Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

doc. Mgr. Jiří Dvorský, Ph.D.

Stav prezentace ke dni 15. února 2025

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Osnova přednášky

Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

- Co je to algoritmus?
- Základy algoritmického řešení problémů
- Důležité typy problémů
- Základní datové struktury
  - Lineární datové struktury
  - Grafy
  - Stromy
  - Množiny a slovníky

Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

Co je to algoritmus?

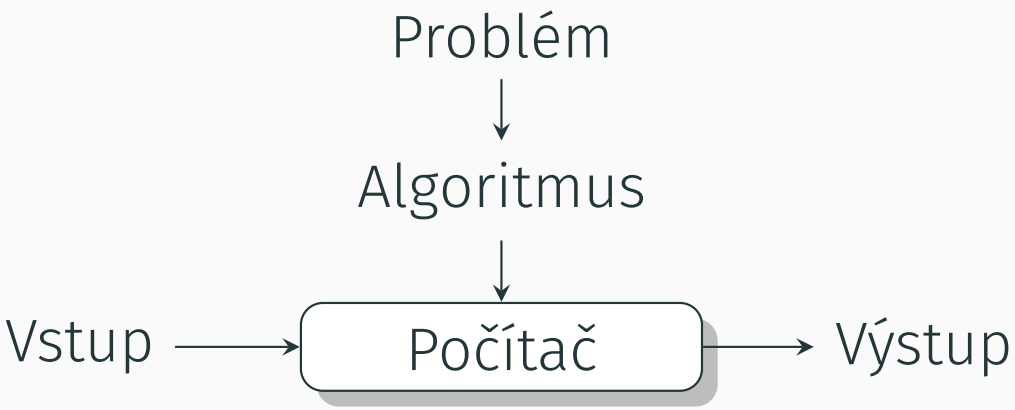
Proč studovat algoritmy?

- Profesionální vývojář/informatik by měl znát standardní algoritmy pro řešení základních problémů, měl by umět navrhovat nové algoritmy a analyzovat efektivitu algoritmů.
- Algoritmy vedou k rozvoji analytického myšlení – jde o nalezení přesného a formálního postupu jak problém řešit.
- Jde o obecně použitelný mentální nástroj – **člověk problému dostatečně nerozumí do té doby, dokud ho nedokáže vysvětlit někomu jinému, neřkuli vysvětlit ho počítači**.
- Schopnost formalizovat řešení vede k daleko hlubšímu pochopení problematiky, než kdybychom se jednoduše pokusili řešit problém řekněme ad-hoc způsobem.

Co je to algoritmus?

Algoritmus

Algoritmus chápeme jako konečnou posloupnost jednoznačných instrukcí vedoucích k řešení problému, tj. vedoucích k získání požadovaného výstupu pro libovolný správný vstup v konečném čase.



Co je to algoritmus? (pokrač.)

- Předchozí popis pojmu algoritmus není definicí v matematickém slova smyslu.
- Předpokládáme, že existuje něco nebo někdo kdo je schopen porozumět „jednoznačným instrukcím“ a je schopen se jimi řídit.
- Pro korektní definici bychom museli nejdříve jasně definovat, co je to ta jednoznačná instrukce.
- Formální definice algoritmu neexistuje!

Co je to algoritmus? (pokrač.)

Poznámky

- Automatický předpoklad – algoritmus bude vykonávat elektronický počítač, computer.
- Překlad slova **computer**:
  1. dnes – počítač
  2. dříve – **počtář**, člověk zapojený do numerických výpočtů.
- Přestože budeme dále předpokládat, že algoritmy budeme implementovat na elektronickém počítači, pojem algoritmu samotný na elektronických počítačích nezávisí.

Ukázka algoritmu

- Tři algoritmy pro řešení téhož problému – nalezení největšího společného dělitele dvou celých čísel.
- Demonstrace několika důležitých skutečností:
  - dodržení požadavku jednoznačnosti instrukcí,
  - rozsah vstupních hodnot je nutné přesně specifikovat,
  - ten samý algoritmus můžeme reprezentovat několika různými způsoby,
  - pro řešení jednoho problému může existovat více algoritmů a
  - algoritmy řešící stejný problém mohou být založeny na zcela rozdílných myšlenkách, principech a mohou se výrazně lišit v rychlosti řešení daného problému.

Největší společný dělitel (NSD, GCD)

- Mějme dvě nezáporná celá čísla m a n, z nichž aspoň jedno je navíc různé od nuly.
- Největší společný dělitel $\gcd(m, n)$ definujeme jako největší celé číslo, které dělí obě čísla m a n beze zbytku.
- Algoritmus pro jeho nalezení popsal v knize „Základy“ Euklides z Alexandrie zhruba ve třetím století před naším letopočtem.

NSD – Euklidův algoritmus

Algoritmus je založen na opakovaném aplikování vztahu

$$\gcd(m, n) = \gcd(n, m \bmod n), \tag{1}$$

dokud zbytek $m \bmod n$ není roven 0.

Protože $\gcd(m, 0) = m$, je poslední hodnota m rovna hledanému největšímu společnému děliteli.

Příklad

$$
\begin{aligned}
\gcd(60, 24) &= \gcd(24, 12) = \gcd(12, 0) = 12 \\
\gcd(24, 60) &= \gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12 \\
\gcd(7, 3) &= \gcd(3, 1) = \gcd(1, 1) = \gcd(1, 0) = 1 \\
\gcd(3, 7) &= \gcd(7, 3) = \gcd(3, 1) = \gcd(1, 1) = \gcd(1, 0) = 1 \\
\gcd(13, 0) &= 13 \\
\gcd(0, 13) &= \gcd(13, 0) = 13
\end{aligned}
$$

NSD – Euklidův algoritmus (pokrač.)

Strukturovanější forma zápisu – jednotlivé kroky výpočtu:

Krok 1 Jestliže $n = 0$ potom vrať hodnotu m jako výsledek a skonči; jinak pokračuj Krokem 2.

Krok 2 Vyděl číslo m číslem n, zbytek po dělení přiřaď do r.

Krok 3 Přiřaď hodnotu čísla n do m, hodnotu čísla r do n. Pokračuj Krokem 1.

NSD – Euklidův algoritmus (pokrač.)

Zápis pomocí pseudokódu:

Vstup : Dvě nezáporná celá čísla m a n, z nichž aspoň jedno je nenulové

Výstup: Největší společný dělitel čísel m a n, $\gcd(m, n)$

```
while n ≠ 0 do
    r ← m mod n;
    m ← n;
    n ← r;
end
return m;
```

NSD – algoritmus postupným dělením

- Algoritmus založen přímo na definici NSD – NSD dělí obě zadaná čísla m a n beze zbytku.
- NSD nemůže být větší než menší ze zadaných čísel, takže můžeme psát $t = \min(m, n)$.
- Pokud t dělí obě čísla m a n beze zbytku, pak $\gcd(m, n) = t$, jinak číslo t snížíme o 1 a postup opakujeme.
- Kdy se algoritmus zastaví?

Příklad

Pro $m = 60$ a $n = 24$ je $t = \min(60, 24) = 24$.
Algoritmus nejprve vyzkouší $t = 24$, pak $t = 23$ a tak dále až se nakonec zastaví na čísle $t = 12$.

NSD – algoritmus postupným dělením (pokrač.)

Strukturovanější forma zápisu – jednotlivé kroky výpočtu:

Krok 1 Přiřaď do t hodnotu $\min(m, n)$.

Krok 2 Vyděl číslo m číslem t. Jestliže zbytek po dělení je roven 0, pokračuj Krokem 3; jinak pokračuj Krokem 4.

Krok 3 Vyděl číslo n číslem t. Jestliže je zbytek po dělení roven 0, vrať číslo t jako výsledek a skonči; jinak pokračuj Krokem 4.

Krok 4 Sniž hodnotu čísla t o 1 a pokračuj Krokem 2.

NSD – algoritmus postupným dělením (pokrač.)

Chyba v algoritmu

- Algoritmus v této podobě nefunguje správně, pokud je jedno z čísel m a n rovno 0. Číslo t by mělo hodnotu 0 a došlo by k dělení nulou.
- **Požadavky na hodnoty vstupující do algoritmu je nutno pečlivě specifikovat!**

NSD – algoritmus rozkladem na prvočinitele

Krok 1 Proveď rozklad čísla m na prvočinitele.

Krok 2 Proveď rozklad čísla n na prvočinitele.

Krok 3 Najdi všechny společné prvočinitele v rozkladech získaných v Kroku 1 a Kroku 2.
Počet výskytů společného prvočinitele p je roven

$$\min(p_m, p_n),$$

kde p_m resp. p_n je počet výskytů p v rozkladu čísla m resp. n,

Krok 4 Spočítej součin všech společných prvočinitelů a tento součin vrať jako výsledek.

NSD – algoritmus rozkladem na prvočinitele (pokrač.)

Příklad

Pro $m = 60$ a $n = 24$ bude průběh algoritmu následovný:

$$\begin{aligned} 60 &= 2^2 \cdot 3^1 \cdot 5^1 \\ 24 &= 2^3 \cdot 3^1 \\ \gcd(60, 24) &= 2^2 \cdot 3^1 \\ &= 12 \end{aligned}$$

NSD – algoritmus rozkladem na prvočinitele (pokrač.)

Problémy

- Popsaný algoritmus je výpočetně mnohem náročnější než Euklidův algoritmus.
- Nalezení NSD pomocí rozkladu na prvočinitele není algoritmus – rozklad čísla není „jednoznačná instrukce“.
- Pro rozklad na prvočinitele je totiž nezbytný seznam prvočísel.
- Krok 3 také není zřejmý – jak nalézt společné prvky v prvočíselném rozkladu? Jak nalézt společné prvky ve dvou setříděných seznamech čísel?

Eratosthenovo síto

- Řešení problému nalezení všech prvočísel menších nebo rovných číslu n, kde $n > 1$.
- Původ v Řecku, cca 200 let před naším letopočtem.
- Nejprve si vytvoříme seznam všech přirozených čísel od 2 do n.
- Postupně bereme čísla, které zůstávají v seznamu a vylučujeme jeho násobky.
- Tímto způsobem pokračujeme dokud, nelze ze seznamu vyloučit žádné další číslo.
- Čísla, která zůstala v seznamu jsou hledaná prvočísla.

Eratosthenovo síto (pokrač.)

Příklad

Pro $n = 25$ dostáváme

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Eratosthenovo síto (pokrač.)

Zastavení algoritmu

- V příkladu jsme jako poslední vylučovali násobky čísla 5.
- Jaké bude, pro dané n, největší číslo p jehož násobky budeme ze seznamu vylučovat?
- První násobek bude $p \cdot p$, tj. p^2.
- Všechny nižší násobky $2p, 3p, \ldots, (p-1)p$ již byly eliminovány jako násobky jiných čísel: $2p$ jako násobek 2, $3p$ jako násobek 3 a tak dále.
- Dále je zřejmé, že $p^2 \leq n$ a tudíž $p = \lfloor \sqrt{n} \rfloor$, kde $\lfloor x \rfloor$ značí nejbližší menší přirozené číslo k číslu x.

Eratosthenovo síto (pokrač.)

Vstup : Přirozené číslo $n > 1$

Výstup : Pole L obsahující všechna prvočísla $\leq n$

```
for p ← 2 to n do
    A[p] ← p;
end
for p ← 2 to ⌊√n⌋ do
    if A[p] ≠ 0 then            // p nebylo dosud vyloučeno
        j ← p²;
        while j ≤ n do
            A[j] ← 0;
            j ← j + p;
        end
    end
end
```

Eratosthenovo síto (pokrač.)

```
// Čísla, která nebyla z pole A vyloučena, okopírujeme
   do pole L
i ← 0;
for p ← 2 to n do
    if A[p] ≠ 0 then
        L[i] ← A[p];
        i ← i + 1;
    end
end
```

NSD – algoritmus rozkladem na prvočinitele

- Začleněním Eratosthenova síta dostáváme pro výpočet největšího společného dělitele pomocí prvočíselného rozkladu regulérní algoritmus.
- Zbývá vyřešit problém, kdy jedno nebo obě čísla, pro než počítáme největšího společného dělitele, je rovno 1…

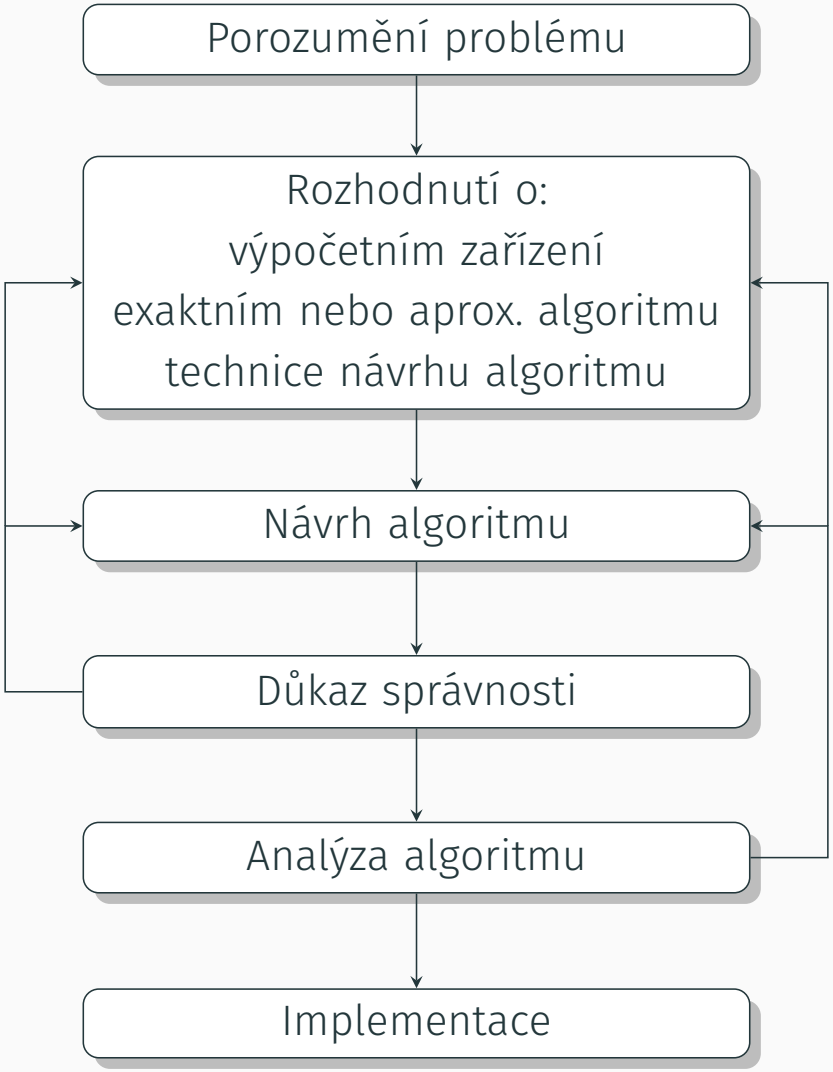
Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

Základy algoritmického řešení problémů

Základy algoritmického řešení problémů

- Algoritmy považujeme za **procedurální, konstruktivní** způsob řešení daného problému.
- Algoritmy nejsou řešením problému samy o sobě, ale jsou návodem jak řešení získat.
- Informatika vs. matematika – neexistence „nekonečně malého ε“, „limity pro n jdoucího k nekonečnu“.
- Podobnost informatiky a starořeckého pojetí geometrie – řešení pomocí „pravítka a kružítka“, konečný počet kroků.

Proces návrhu a analýzy algoritmu



Porozumění problému

- Na první pohled banalita – nesprávné porozumění se může vymstít ⇒ nutnost algoritmus přepracovat.
- Řešení ukázkových případů, speciální případy řešení.
- Vstupní data definují **instanci problému**. Definice přípustných vstupních dat.
- **Správný algoritmus** musí **pracovat správně** pro **všechna** přípustná vstupní data, ne jen pro **většinu**.
- Znalost odborné literatury výhodou – typické problémy a jejich typická řešení.
- Není tedy vždy nutné „vynalézat znovu kolo“.
- Pro výběr vhodného algoritmu je dobré znát jejich silné i slabé stránky.

Výpočetní zařízení

- Výpočetní zařízení – počítač nemusí být jen „notebook“.
- Paralelní výpočetní zařízení – vícejádrové procesory, akcelerátory CUDA, paralelní superpočítače.
- Dosud převažuje **von Neumannova architektura** (John von Neumann 1946).
- V dalším výkladu se budeme zabývat **sekvenčními algoritmy** na von Neumannově architektuře.
- **Random Access Machine** (RAM) – teoretický model von Neumannovy architektury počítače.

Výpočetní zařízení (pokrač.)

- Pro návrh algoritmu a zkoumání jeho efektivity je vhodné využít RAM – HW a SW nezávislost.
- Praktická implementace – je nutné brát v úvahu HW a SW omezení konkrétního počítače.
- Předpoklad dostatečného výkonu použitého počítače. Počítačový „pravěk“.

Výpočetní zařízení (pokrač.)

Varování

„The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times;

premature optimization is the root of all evil

(or at least most of it) in programming.“

Donald Knuth, The Art of Computer Programming

Přesné vs. přibližné řešení problému

- **Exaktní algoritmus** – poskytuje přesné řešení.
- **Aproximační algoritmus** – poskytuje přibližné řešení.
  Využití aproximačních algoritmů:
  1. Existují důležité problémy, které neumíme přesně řešit, např. aerodynamické a hydrodynamické problémy.
  2. Exaktní algoritmy jsou z podstaty problému nepřijatelně pomalé. Pomalost je způsobena obrovským počet možných řešení, ne nekvalitním algoritmem nebo implementací.
  3. Aproximační algoritmus je součástí sofistikovaného exaktního algoritmu.

Poznámka

Pokud nepotřebujeme striktně oddělit algoritmus pro přesné a pro přibližné řešení problému, přívlastek exaktní obvykle vynecháváme.

Techniky návrhu algoritmů

- Máme pohromadě vše potřebné: pochopili jsme zadaný problém, zvolili výpočetní zařízení a rozhodli zda použijeme exaktní nebo aproximační algoritmus.
- Jak postupovat při návrhu algoritmu? Jakou použít techniku návrhu algoritmu?

Definice

Technika návrhu algoritmu (strategie návrhu algoritmu či paradigma návrhu algoritmu) je obecný přístup k algoritmickému řešení problémů, který je aplikovatelný na množství problémů z různých oblastí informatiky.

Techniky návrhu algoritmů (pokrač.)

Přínosnost technik návrhu algoritmů

1. poskytují návod, jak navrhovat algoritmy pro nové problémy, pro které není znám uspokojivý algoritmus, a
2. umožňují přehledně klasifikovat nejrůznější algoritmy podle jejich základní myšlenky.

Techniky návrhu algoritmů (pokrač.)

Mějme však na paměti, že

- návrh konkrétního algoritmu pro řešení konkrétního problému může být velice náročným úkolem,
- ne všechny techniky návrhu algoritmu lze aplikovat na konkrétní problém, někdy je nutné techniky kombinovat,
- může být obtížné rozpoznat na jaké technice návrhu je algoritmus založen,
- i pokud je technika jasná, sestavení algoritmu často vyžaduje netriviální úsilí a vynalézavost, avšak
- s přibývající vývojářovou praxí se vše stává snazší a snazší, ale zřídkakdy snadné.

Techniky návrhu algoritmů (pokrač.)

Význam datových struktur

- vhodná datová struktura má zásadní význam pro navrhovaný algoritmus – Eratosthenovo síto versus spojový seznam
- některé z technik návrhu algoritmů silně závisí na struktuře nebo na restrukturalizaci dat určujících instanci řešeného problému,
- Niklaus Wirth: „Algorithms + Data Structures = Programs“

Metody zápisu algoritmu při jeho návrhu

Přirozený jazyk

- nemusí jít o písemný záznam – ústně formulovaná myšlenka
- možné nejednoznačnosti – extrémní případ „Ženu holí stroj.“
- schopnost precizně formulovat myšlenky, formulovat je logicky správně, definovat pojmy popisujících problém, pojmy zařazovat do myšlenkového schématu atd.

Metody zápisu algoritmu při jeho návrhu (pokrač.)

Pseudokód

- směs přirozeného jazyka a konstrukcí podobných programovacím jazykům.
- obvykle přesnější a stručnější než přirozený jazyk
- stručnější zápis navrhovaného algoritmu
- existuje množství navzájem podobných „dialektů“ pseudokódu

Metody zápisu algoritmu při jeho návrhu (pokrač.)

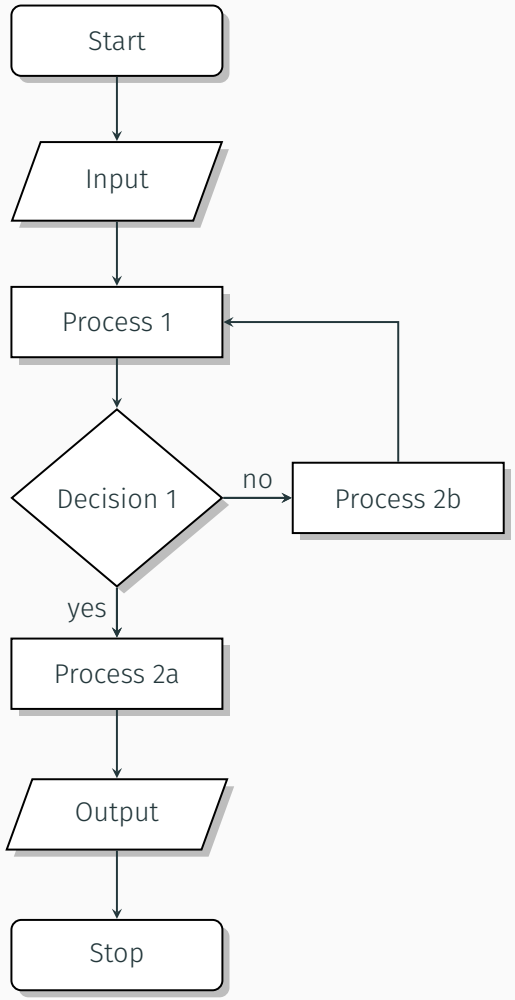
Programovací jazyk

- další možný způsob zápisu
- tento zápis považován spíše za implementaci

Metody zápisu algoritmu při jeho návrhu (pokrač.)

Vývojový diagram

- angl. flowchart
- grafická forma zápisu algoritmu
- dnes již nepoužíváno



Důkaz správnosti algoritmu

Definice

Algoritmus považujeme za **správný**, pokud pro každý správný vstup poskytne v konečném čase správný výsledek. Pro nesprávný vstup není chování správného algoritmu definováno.

- Obvyklou metodou důkazu je matematická indukce.
- Důkaz správnosti vs. nesprávnosti algoritmu
  - Pro korektní důkaz správnosti algoritmu nestačí prokázat správnost pro **některou** instanci problému, správnost musíme umět prokázat pro **všechny** instance problému, a naopak

Důkaz správnosti algoritmu (pokrač.)

  - jako důkaz nesprávnosti algoritmu stačí najít **jedinou** instanci problému, abychom mohli algoritmus prohlásit za chybný.
- Správnost aproximačního algoritmu – chyba výsledku algoritmu, nepřesáhne předem definovanou mez.

Analýza algoritmu – zkoumané vlastnosti

Správnost – již vyřešeno

Časová složitost (angl. **time complexity**)

- „jak rychle algoritmus pracuje“
- rychlost neměříme časovými jednotkami, ale množstvím provedených instrukcí algoritmu (stejný algoritmus na rychlejším a pomalejším HW)

Prostorová složitost (angl. **space complexity**)

- „jak mnoho paměti algoritmus potřebuje“
- měříme v bytech a násobcích

Analýza algoritmu – zkoumané vlastnosti (pokrač.)

Jednoduchost (angl. **simplicity**)

- nelze exaktně definovat, na rozdíl od složitosti,
- spíše jde o subjektivní záležitost – krása, elegance (NSD Euklidovým algoritmem vs. rozklad na prvočinitele),
- jednodušší algoritmus – snazší pochopit, implementovat, patrně i menší množství chyb,
- jednodušší algoritmus – nemusí mít nutně nižší složitost,
- použití – typicky prototyp SW. Pokud nevyhovuje – přechod na algoritmus s nižší složitostí. Ale! „Předčasná optimalizace...“

Analýza algoritmu – zkoumané vlastnosti (pokrač.)

- terminologie – opak jednoduchosti není „složitost“ algoritmu, ale komplikovanost, nesrozumitelnost, nevhodnost návrhu.

Obecnost

1. obecnost navrženého algoritmického řešení – řešit problém velice obecně a nebo brát v úvahu možná zjednodušení v konkrétním případě?
   - řešení obecnějšího problému je lehčí než konkrétního – např. nesoudělnost dvou čísel, řešení přes NSD, NSD je obecnější problém

Analýza algoritmu – zkoumané vlastnosti (pokrač.)

- řešení obecnějšího a konkrétního problému na stejné úrovni – např. hledání mediánu, řešení přes třídění (obecnější) i konkrétní algoritmus
- řešení obecnějšího je výrazně náročnější – např. kvadratická rovnice $ax^2 + bx + c = 0$ versus obecná algebraická rovnice n-tého stupně.

2. obecnost instance problému – návrh algoritmu by měl zpracovat všechny rozumně očekávatelné, přirozené, instance problému.
   - U NSD není přirozené vyloučit číslo 1, ale
   - u kvadratické rovnice většinou předpokládáme, že a, b a c jsou reálná čísla – obecněji lze brát i čísla komplexní.

Analýza algoritmu – zkoumané vlastnosti (pokrač.)

Pokud nejsme spokojeni se složitostí nebo jednoduchostí návrhu či obecností algoritmu?

Nezbývá nic jiného než se vrátit na začátek, sednout si za stůl, vzít do ruky tužku a papír a přemýšlet, kreslit, hledat v literatuře a tak dále.

„Konstruktér ví, kdy dosáhl dokonalosti. Ne v okamžiku, kdy již není co přidat, ale v okamžiku, kdy již není co odebrat.“

Antoine de Saint-Exupéry

„Keep It Simple, Stupid!“

Kelly Johnson

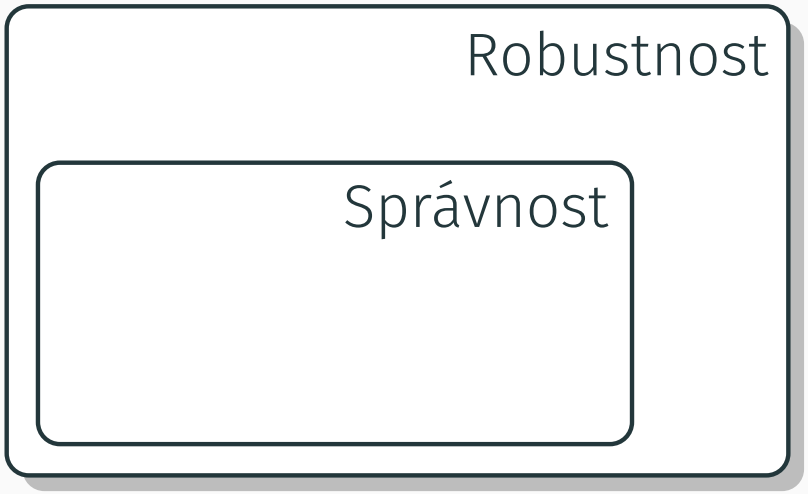
Kódování algoritmu

- Opět podceňovaná fáze – „Algoritmus máme vymyšlený, tak teď to jenom přepíšeme do počítače a máme hotovo.“
- Algoritmus implementujeme buď nesprávně nebo neefektivně nebo dokonce nastanou obě možnosti najednou.
- V praktickém životě – správnost programů je ověřována testováním.
- Testování programů je „umělecké řemeslo“.
- Další kritické místo – vstup dat.
  - Škola – vstupní data definují korektní instanci řešeného problému.
  - Praxe – otázku kontroly vstupních dat je nutno řešit.

Robustnost algoritmu

Definice

Algoritmus považujeme za **robustní**, pokud je správný a pro každý nesprávný vstup vydá hlášení o chybě a je schopen se z chyby zotavit.



Efektivita implementace

- Správnost implementace algoritmu je nezbytnost.
- Ale i správnou implementaci lze provést neefektivně, výkon počítače není využit tak, jak by mohl být.
- Optimalizace kódu:
  1. manuální – výpočet invariantu cyklu, nahrazení společných podvýrazů proměnnou.
  2. automatická – algoritmy optimalizace zabudované do kompilátorů, např. přidělování registrů.
- Optimalizací kódu lze zlepšit efektivitu programu o nějaký konstantní faktor, např. 10%.
- Pro radikální, řádové, zlepšení je nutné implementovat algoritmus s nižší složitostí.

Efektivita implementace (pokrač.)

- Hledání stále lepšího a lepšího algoritmu zajímavé mentální dobrodružství...
- Otázkou je kdy přestat. Dokonalost je drahý luxus. Inženýrský přístup – zdroje alokované pro projekt.
- Akademická otázka **optimality algoritmu**: „Jaká je nejmenší možná složitost jakéhokoliv algoritmu, který vyřeší daný problém?“
- Například sekvenční algoritmus pro třídění pole s n prvky – minimálně $n \log_2 n$ porovnání.
- Lze každý problém řešit algoritmem? Nerozhodnutelné problémy – nelze je řešit jakýmkoliv algoritmem.

Efektivita implementace (pokrač.)

- Naštěstí většinu problémů z praktického života lze algoritmicky řešit.

  Dobrý algoritmus je výsledkem opakovaného úsilí a několikanásobného přepracovávání.

Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

Důležité typy problémů

Důležité typy problémů

- Třídění
- Vyhledávání
- Zpracování řetězců
- Grafové úlohy
- Kombinatorické úlohy
- Geometrické úlohy
- Numerické úlohy

Třídění

- Třídění v informatice – přeuspořádání prvků do neklesající posloupnosti. Srovnej s tříděním odpadu.
- Mezi prvky musí být definována **relace uspořádání** čili vztah „menší nebo rovno“, $\leq$.
- V praxi třídíme čísla, řetězce či strukturované záznamy.
- U záznamu musíme definovat **klíč** tj. část záznamu pole které třídíme pro kterou je definováno uspořádání. Klíč nemusí být definován explicitně, např. u čísel je jím číslo samo.

Uspořádání

Definice

Mějme binární homogenní relaci $\rho \subseteq A \times A$ na množině A.

- Relaci ρ nazýváne (neostré) **částečné uspořádání**, jestliže je současně reflexivní, antisymetrická a tranzitivní.
- Relaci ρ nazýváme (neostré) **úplné uspořádání**, jestliže je současně reflexivní, antisymetrická, tranzitivní a úplná.
- Relaci ρ nazýváme (částečné) **ostré uspořádání**, jestliže je současně asymetrická (a tedy i antisymetrická a ireflexivní) a tranzitivní.
- Relaci ρ nazýváme **úplné ostré uspořádání**, jestliže je současně asymetrická (a tedy i antisymetrická a ireflexivní), tranzitivní a souvislá.

Uspořádání – poznámky

- Neostré uspořádní standardně značíme $\leq$, ostré uspořádání pak $<$.
- Místo označení **částečné uspořádání** používáme někdy jen **uspořádání**.
- Místo termínu **úplné uspořádání** se používá i termín **totální** či **lineární** uspořádání.
- Je-li $\leq$ uspořádání na množině A, pak relačnímu systému $(A, \leq)$ říkáme **uspořádaná množina** (angl. **poset** – partially oredered set). Úplně uspořádané množině říkáme **řetězec** (angl. **chain**).

Uspořádání – poznámky (pokrač.)

- Dva různé prvky x, y jsou **porovnatelné** v uspořádání $\leq$, jestliže platí $(x \leq y) \vee (y \leq x)$. V opačném případě jsou prvky **neporovnatelné**. V úplném uspořádání jsou každé dva prvky porovnatelné.
- Průnik uspořádání je opět uspořádáním. Sjednocení uspořádání nemusí být obecně uspořádáním.
- Vztah mezi ostrým a neostrým uspořádáním je možno zapsat takto: $" \leq " = " < " \cup " = "$, tj. přidáním identické relace („rovnosti“) k ostrému uspořádání.

Použité vlastnosti binárních homogenních relací

Použité vlastnosti relace $\rho \subseteq A \times A$ $\forall x, y, z \in A$:

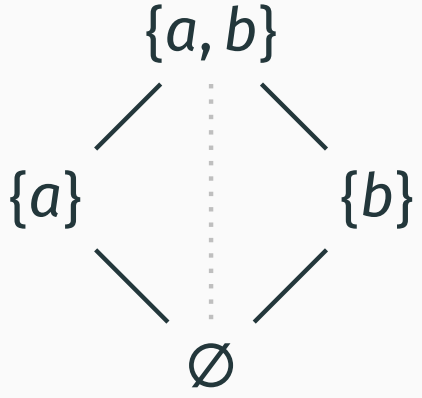
- reflexivita: $x\rho x$,
- ireflexivita: $\neg(x\rho x)$,
- asymetrie: $x\rho y \Rightarrow y\rho x$,
- antisymetrie: $x\rho y \wedge y\rho x \Rightarrow x = y$,
- tranzitivita: $x\rho y \wedge y\rho z \Rightarrow x\rho z$,
- souvislost: $[x \neq y \Rightarrow x\rho y \vee y\rho x]$,
- úplnost: $x\rho y \vee y\rho x$.

Hasseův diagram

Relaci uspořádání standardně znázorňujeme pomocí **Hasseova diagramu**, který

- reprezentuje relaci bezprostředního předcházení bez tranzitivních hran, která je stejná pro ostré i neostré uspořádání a který
- odpovídá orientovanému grafu, kde jsou všechny hrany orientovány zdola nahoru.

Příklad



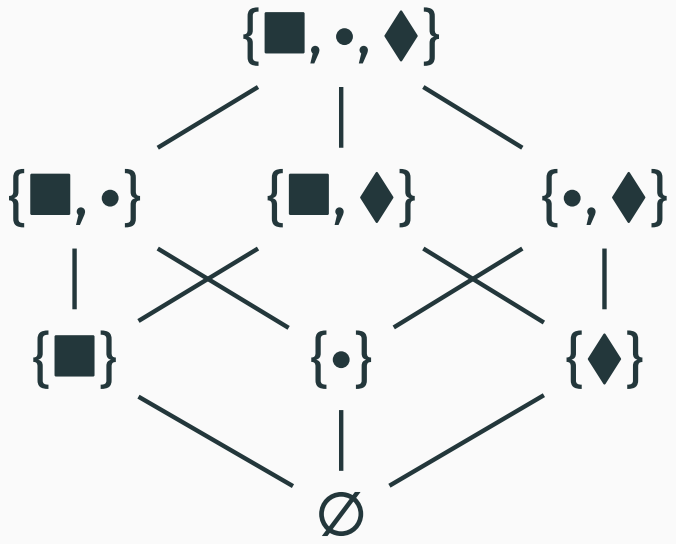
Hasseův diagram pro relaci uspořádání „být podmnožinou“ na množině $\{a, b\}$. Tranzitivní hrana, která se běžně nezobrazuje, je zobrazena tečkovaně.

Částečné uspořádání – příklad

Pro libovolnou množinu A můžeme definovat na množině jejích podmnožin $P(A)$ uspořádání $\leq$ inkluzí: $X \leq Y$ pokud $X \subseteq Y$, kde $X, Y \in P(A)$.

Takto definované uspořádání není úplné ale pouze částečné, protože v něm existují neporovnatelné prvky.

Příklad



V $A = \{■, •, ♦\}$ jsou neporovnatelnými prvky

- všechny jednoprvkové podmnožiny mezi sebou a
- všechny dvouprvkové podmnožiny mezi sebou.

Úplné uspořádání – příklad

- Obvyklá relace **<** na množině přirozených, celých, racionálních a reálných číslech je úplné uspořádání.
- Abecední, lexikografické, uspořádání řetězců je také úplné uspořádání.
- Správně seskládané matrjošky jsou úplně uspořádané pomocí relace „být uvnitř“. Ale pouze za předpokladu, že do žádné bábušky se nesmí vejít více menších vedle sebe – v tom případě dostáváme pouze částečné uspořádání.

Třídění – využití

- Setříděný seznam hodnot je požadovaným výstupem – výsledková listina závodu, výsledky vyhledávání na internetu.
- Pro některé úlohy se řeší lépe pro setříděný vstup – typicky **vyhledávání**. Telefonní seznam. Geometrické úlohy. Komprese dat. Hladové algoritmy.

Definice třídícího problému

- Předpokládejme posloupnost prvků $A = a_1, a_2, \dots a_n$. Úkolem třídění je nalézt permutaci $\pi : \mathbb{N} \to \mathbb{N}$ takovou, že $a_{\pi_i} \leq a_{\pi_{i+1}}$ pro všechna $1 \leq i < n$.
- Permutaci π nelze nalézt přímo, permutací n prvků je $n!$.
- Třídící algoritmy budeme chápat jako algoritmy, které postupně konstruují permutaci π, například porovnáváním a výměnou prvků.

Definice třídícího problému – příklad

Mějme posloupnost $A = ebfcda$ a obvyklé abecední uspořádání písmen. Hledaná permutace je

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 4 & 5 & 1 & 3 \end{pmatrix}$$

Potom

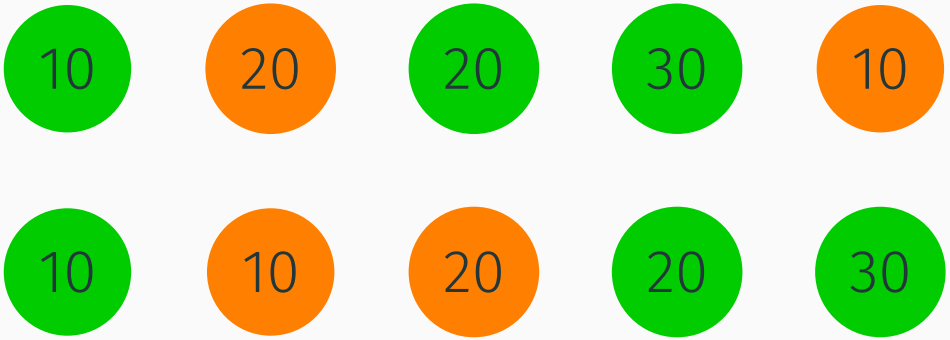
$$a_{\pi_1} < a_{\pi_2} < a_{\pi_3} < a_{\pi_4} < a_{\pi_5} < a_{\pi_6}$$
$$a_6 < a_2 < a_4 < a_5 < a_1 < a_3$$
$$a < b < c < d < e < f$$

Třídění

- Třídících algoritmů byla vyvinuta celá řada. Neexistuje jeden univerzální algoritmus pro všechny situace.
  - jednoduché a pomalé vs. komplexní a rychlé,
  - náhodná vs. téměř setříděná posloupnost na vstupu
  - vnitřní paměť vs. vnější paměť.
- Máme-li dáno n prvků, minimální počet porovnání je $n \log_2 n$ pro sériové algoritmy založené na porovnání a výměně.

Třídění (pokrač.)

- **Stabilní třídění** – zachovává vzájemné polohy prvků. Jestliže máme ve tříděné posloupnosti dva prvky se shodným klíčem na pozici i a j, kde $i < j$, pak po setřídění budou tyto prvky na pozicích i' a j', kde $i' < j'$.



Nápověda: sledujte vzájemné polohy oranžových a zelených čísel.

Algoritmy třídící pomocí výměn na velkou vzdálenost jsou obvykle rychlejší, ale zase nejsou stabilní.

Třídění (pokrač.)

- Třídění **in-situ** – třídící algoritmus si vystačí jen se pamětí pro uložení prvků plus přídavná paměť konstantního rozsahu tj. tato paměť nezávisí na počtu tříděným prvků, typicky proměnné pro průchod cyklem, logické příznaky atd.
- **Přirozené** třídění – složitost třídícího algoritmu roste s mírou nesetříděnosti vstupních dat.

Míra nestříděnosti posloupnosti dat

- Cílem je najít měřítko nesetříděnosti, „rozházenosti“, posloupnosti n prvků, které máme třídit.
- Setříděné posloupnosti by měla odpovídat **nulová** nesetříděnost.
- Posloupnosti setříděné v opačném pořadí by měla odpovídat **maximální** nesetříděnost.
- Ostatní posloupnosti by se měly pohybovat mezi těmito krajními možnostmi

Míra nestříděnosti permutace

- Permutace čísel $1 \dots n$.
- Identická permutace – nulová nesetříděnost

$$\pi_{id} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

- Obrácená permutace – maximální nesetříděnost

$$\pi_{rev} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

- Nesetříděnost permutace budeme měřit **počtem inverzí** dané permutace.

Inverze v permutaci

Definice

Mějme permutaci $\pi : \mathbb{N} \to \mathbb{N}$. Inverze v permutaci π je dvojice prvků i, j taková, že $i < j$ a zároveň $\pi_i > \pi_j$.

Inverzi v permutaci můžeme volně interpretovat tak, že „na menším indexu je větší prvek a současně na větším indexu je menší prvek“.

Příklad

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$$

Všechny permutace v π

$1 < 2 \wedge 4 > 1$ $\quad$ $1 < 4 \wedge 4 > 2$

$1 < 3 \wedge 4 > 3$ $\quad$ $3 < 4 \wedge 3 > 2$

Počet inverzí v permutaci

- Identická permutace – celkový počet inverzí je nulový
- Obrácená permutace

Prvek	Inverze s prvky	Počet inverzí
n	$n-1, n-2, n-3, \dots, 1$	$n-1$
$n-1$	$n-2, n-3, \dots, 1$	$n-2$
$\vdots$	$\vdots$	$\vdots$
3	2, 1	2
2	1	1
1	–	0

Celkový počet inverzí je roven

$$(n-1) + (n-2) + \cdots + 2 + 1 + 0 = \frac{1}{2}n(n-1)$$

Průměrný počet inverzí v permutaci

- Označme C_n celkový počet inverzí ve všech permutacích n prvků. Nejprve odvodíme vztah mezi C_n a C_{n-1}.
- Uvažujme všechny permutace $n - 1$ prvků. Ke všem těmto permutacím přidáme n za poslední prvek permutace. Počet inverzí se nezvýší a bude roven C_{n-1}.
- K permutacím $n - 1$ prvků přidáme n za předposlední prvek permutace. Počet inverzí se zvýší o jednu pro každou permutaci, tedy $C_{n-1} + 1 \cdot (n - 1)!$
- Až nakonec k permutacím $n - 1$ prvků přidáme n před první prvek permutace. Počet inverzí se zvýší o $n - 1$ pro každou permutaci, tedy $C_{n-1} + (n - 1)(n - 1)!$

Průměrný počet inverzí v permutaci (pokrač.)

Proto

$$
\begin{array}{rcccccc}
C_n & = & C_{n-1} & + & 0 \cdot (n-1)! & + \\
 & & C_{n-1} & + & 1 \cdot (n-1)! & + \\
 & & C_{n-1} & + & 2 \cdot (n-1)! & + \\
 & & \vdots & & \vdots & \\
 & & C_{n-1} & + & (n-1)(n-1)! &
\end{array}
$$

Odtud

$$
\begin{aligned}
C_n &= nC_{n-1} + \left[0 + 1 + \cdots + (n-1)\right](n-1)! \\
&= nC_{n-1} + \left[\frac{1}{2}n(n-1)\right](n-1)! \\
&= nC_{n-1} + \frac{1}{2}(n-1)n!
\end{aligned}
$$

Průměrný počet inverzí v permutaci (pokrač.)

Průměrný počet inverzí I_n je roven

$$I_n = \frac{C_n}{n!}.$$

Odtud dosadíme $C_n = n!I_n$ resp. $C_{n-1} = (n-1)!I_{n-1}$ a dostáváme

$$\begin{aligned} n!I_n &= n(n-1)!I_{n-1} + \frac{1}{2}(n-1)n! \\ &= n!I_{n-1} + \frac{1}{2}(n-1)n! \end{aligned}$$

Po vykrácení $n!$ dostáváme

$$I_n = I_{n-1} + \frac{1}{2}(n-1)$$

Průměrný počet inverzí v permutaci (pokrač.)

Rozepíšeme vztah pro I_n

$$\begin{aligned} I_n &= I_{n-2} + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\ &= I_{n-3} + \frac{1}{2}(n-3) + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\ &\vdots \\ &= I_{n-i} + \frac{1}{2}(n-i) + \cdots + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \end{aligned}$$

Dále víme, že jednoprvková permutace nemůže mít inverzi, tudíž $I_1 = 0$.

Průměrný počet inverzí v permutaci (pokrač.)

Nyní hledáme takové i, aby výraz $n - i$ v indexu I_{n-i} byl roven 1. Zřejmě $i = n - 1$ a proto

$$
\begin{aligned}
I_n &= I_{n-(n-1)} + \frac{1}{2}\big[n - (n-1)\big] + \frac{1}{2}\big[n - (n-2)\big] + \cdots + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\
&= I_1 + \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 + \cdots + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\
&= I_1 + \frac{1}{2}\big[1 + 2 + \cdots + (n-2) + (n-1)\big] \\
&= I_1 + \frac{1}{2}\left[\frac{1}{2}n(n-1)\right] \\
&= I_1 + \frac{1}{4}n(n-1)
\end{aligned}
$$

Průměrný počet inverzí v permutaci (pokrač.)

A protože $I_1 = 0$ dostáváme konečně

$$I_n = \frac{1}{4}n(n-1)$$

Shrnutí – počet inverzí v permutaci n prvků

Minimum	0
Průměr	$\frac{1}{4}n(n-1)$
Maximum	$\frac{1}{2}n(n-1)$

Vyhledávání

- Základní úloha – nalezení prvku a v dané množině M, či multimnožině.
- Matematicky – platí $a \in M$ resp. $a \notin M$?
- Matematika neřeší složitost této operace.
- Algoritmů pro vyhledávání existuje celá řada – sekvenční, půlením intervalu, hašování…
- Neexistuje optimální algoritmus pro všechny situace, algoritmy mají různé předpoklady – více paměti pro rychlejší práci, setříděné pole…
- Důležité aspekty:
  - vzájemný poměr operací vyhledávání, vkládání a mazání prvku z množiny – převažuje vyhledávání nebo je poměr vyrovnaný?
  - organizace velmi velkých dat.

Zpracování řetězců

- Řetězec – posloupnost znaků z dané abecedy.
- Typické příklady řetězců:
  - textové řetězce, abeceda složena z písmen, číslic a interpunkce,
  - bitové řetězce složené z 0 a 1 nebo
  - genové řetězce složené ze znaků A, C, G a T
- Využití
  - zpracování textů,
  - komprese dat,
  - programovací jazyky a kompilátory nebo
  - vyhledávání v řetězcích (pattern matching) – hledání jednoho řetězce, vzorku, či vzorků v jiném řetězci. Triviální příklad – `Ctrl+F` v textovém editoru.

Vyhledávání v textu – Pattern Matching

- Při **vyhledávání v textu** zjišťujeme, zda daný **vzorek/vzorky** (pattern) odpovídá, shoduje se, s částí v daného **textu**. Můžeme také říci, že hledáme **výskyty vzorku v textu**.
- Využití:
  - v textových editorech (pohyb v editovaném textu, záměna řetězců),
  - v utilitách typu `grep`, které umožní najít všechny výskyty zadaných vzorků v množině textových souborů,
  - vyhledávání na webu,
  - při zkoumání DNA,
  - při analýze obrazu, zvuku apod.

Vyhledávání v textu – klasifikace vyhledávacích algoritmů

		Předzpracování textu	
		ne	ano
Předzpracování vzorku	ne	vyhledávání hrubou silou	indexové metody, typicky webové vyhledávače, obecně jsou to tzv. Information Retrieval Systems
	ano	pokročilé vyhledávací algoritmy	signaturové vyhledávací metody

Vyhledávání v textu – další kritéria rozdělení

Počet hledaných vzorků – jeden, konečný počet nebo nekonečný počet vzorků

Počet výskytů – první výskyt, všechny výskyty

Způsob porovnávání – přesné vyhledávání versus přibližné vyhledávání, kdy jsou povoleny odchylky mezi vzorkem a textem např. jeden znak se může lišit

Směr vyhledávání – v textu obvykle postupujeme od nižších indexů k vyšším, „zleva doprava“

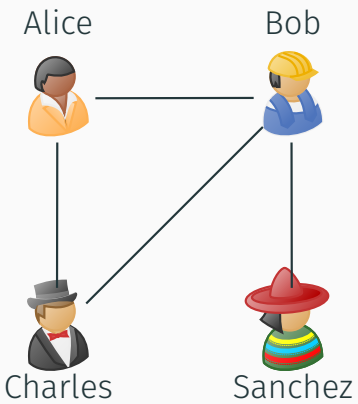
- sousměrné algoritmy – vzorek je procházen stejným směrem
- protisměrné algoritmy – vzorek je procházen opačným směrem.

Vyhledávání v textu – označení

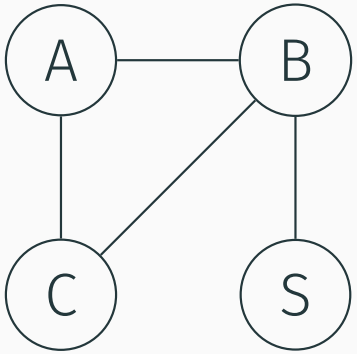
V dalším textu budeme používat následující označení:

- p hledaný vzorek, $p = p_0 p_1 \dots p_{m-1}$, kde $|p| = m$ je délka vzorku,
- t prohledávaný text, $t = t_0 t_1 \dots t_{n-1}$, kde $|t| = n$ je délka vzorku,
- Σ – abeceda z níž je sestaven vzorek i text,
- σ – velikost abecedy Σ ($\sigma = |\Sigma|$),
- $\bar{C}_n$ – očekávaný počet porovnání potřebných k vyhledání vzorku v textu délky n.

Grafové úlohy



Převzato z [1]



- **Graf** – neformálně množina bodů, **vrcholů**, z nichž některé jsou propojeny úsečkami, **hranami**.
- Využití – reprezentace dopravních sítí, projektového řízení, sociální sítě, elektrické sítě atd.
- Základní úlohy:
  - průchod grafem – lze se dostat do všech vrcholů grafu
  - nejkratší cesta – nejkratší cesta mezi dvěma městy

Grafové úlohy (pokrač.)

  - topologické třídění – organizace projektu, činnosti musí na sebe navazovat, lze něco dělat souběžně?
- Výpočetně složité úlohy
  - **Problém obchodního cestujícího** (traveling salesman problem, TSP) – úkolem je najít nejkratší cestu mezi n městy, přičemž každé lze navštívit právě jednou. Logistika, výroba mikročipů.
  - **Problém barvení grafu** – úkolem je najít nejmenší počet barev vrcholů tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu. Plánování – události odpovídají vrcholů, hrany spojují události, které nelze vykonávat současně, řešení problému barvení grafu poskytuje optimální rozvrh.

Kombinatorické úlohy

- Podstata úloh – nalézt **permutaci**, **kombinaci** či **podmnožinu** z dané množiny objektů, která splňuje určitá **omezení** a případně mají nějakou další vlastnost např. minimalizace resp. maximalizace nějaké funkce.
- Problém obchodního cestujícího – pořadí navštívených měst je permutace, minimalizovaná funkce je celková vzdálenost.
- Patrně nejsložitější problémy informatiky z teoretického i praktického pohledu:
  - počet možných kandidátů řešení (např. permutací) roste velice rychle a dosahuje obrovských hodnot už pro středně velké problémy

Kombinatorické úlohy (pokrač.)

  - není znám algoritmus pro nalezení přesného řešení v přijatelném čase a
  - dokonce se neví jestli takový algoritmus existuje, předpokládá se že ne.
- Některé kombinatorické problémy ale **lze** řešit efektivně – např. hledání nejkratší cesty.

Geometrické úlohy

- Zpracovávají body, úsečky, mnohoúhelníky a podobné objekty.
- Jsou to vlastně první algoritmy – euklidovská geometrie, konstrukce „pravítkem a kružítkem“.
- Využití:
  - počítačová grafika,
  - počítačové hry,
  - robotika,
  - medicína.
- V našem předmětu:
  - problém nejbližší dvojice bodů – množina bodů v rovině, najít dva body s minimální vzdáleností,
  - konvexní obal množiny bodů – nalézt nejmenší konvexní mnohoúhelník obsahující dané body.

Numerické úlohy

- Řešení soustav rovnic, výpočet hodnot funkcí, určitých integrálů atd.
- Většina těchto úloh vyžaduje počítání s reálnými čísly. Typické problémy:
  - počítač umí zachytit jen omezený rozsah čísel (ne ∞) a s omezenou přesností ($\frac{1}{3}, \pi$) a
  - kumulace zaokrouhlovacích chyb.
- Vědeckotechnické výpočty – klasická aplikace prvních počítačů. Inženýrské aplikace.
- Dnes – ukládání a analýza dat, navigace, logistika....
- V našem předmětu – několik typických úloh, řešení soustavy rovnic, matice.

Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

Základní datové struktury

Základní datové struktury

- **Datovou strukturu** můžeme definovat jako způsob organizace vzájemně souvisejících dat.
- Jakou použít datovou strukturu silně závisí na řešeném problému.
- Existuje několik zvláště důležitých datových struktur:
  - lineární datové struktury – **pole**, **spojový seznam**, **zásobník**, **fronta**, **prioritní fronta**
  - **graf**
  - **strom**
  - **množina**
  - **slovník**

Pole (Array)

- Konečná posloupnost n hodnot uložených ve spojitém úseku paměti.
- Přístup pomocí indexu, náhodný přístup s konstantní časovou složitostí.
- Index:
  - nezáporné celé číslo,
  - pole s n hodnotami má rozsah indexů vždy $0, \ldots, n - 1$

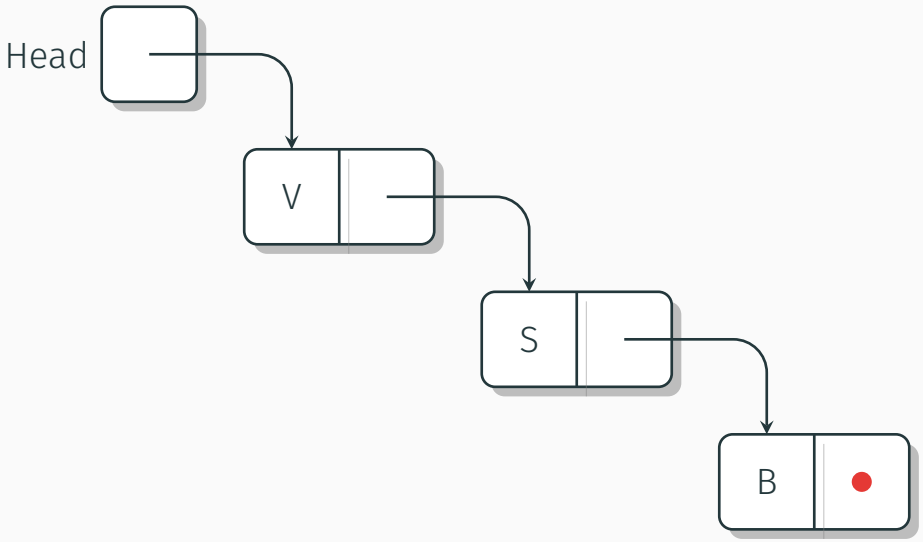
`a[0]`	`a[1]`	···	`a[n-1]`

- Využití:
  - přímo – vektory, buffery,
  - základ pro další datové struktury – řetězce, matice atd.

Seznam (Linked list)

Charakteristika

- nejobecnější lineární datová struktura,
- operace nejsou striktně určeny,
- existuje mnoho variant.



Atributy

- atributy seznamu závisí na konkrétní implementaci,
- v nejjednodušším případě jen nutný odkaz na první prvek seznamu, tzv. **hlavu seznamu**.

Seznam (Linked list) (pokrač.)

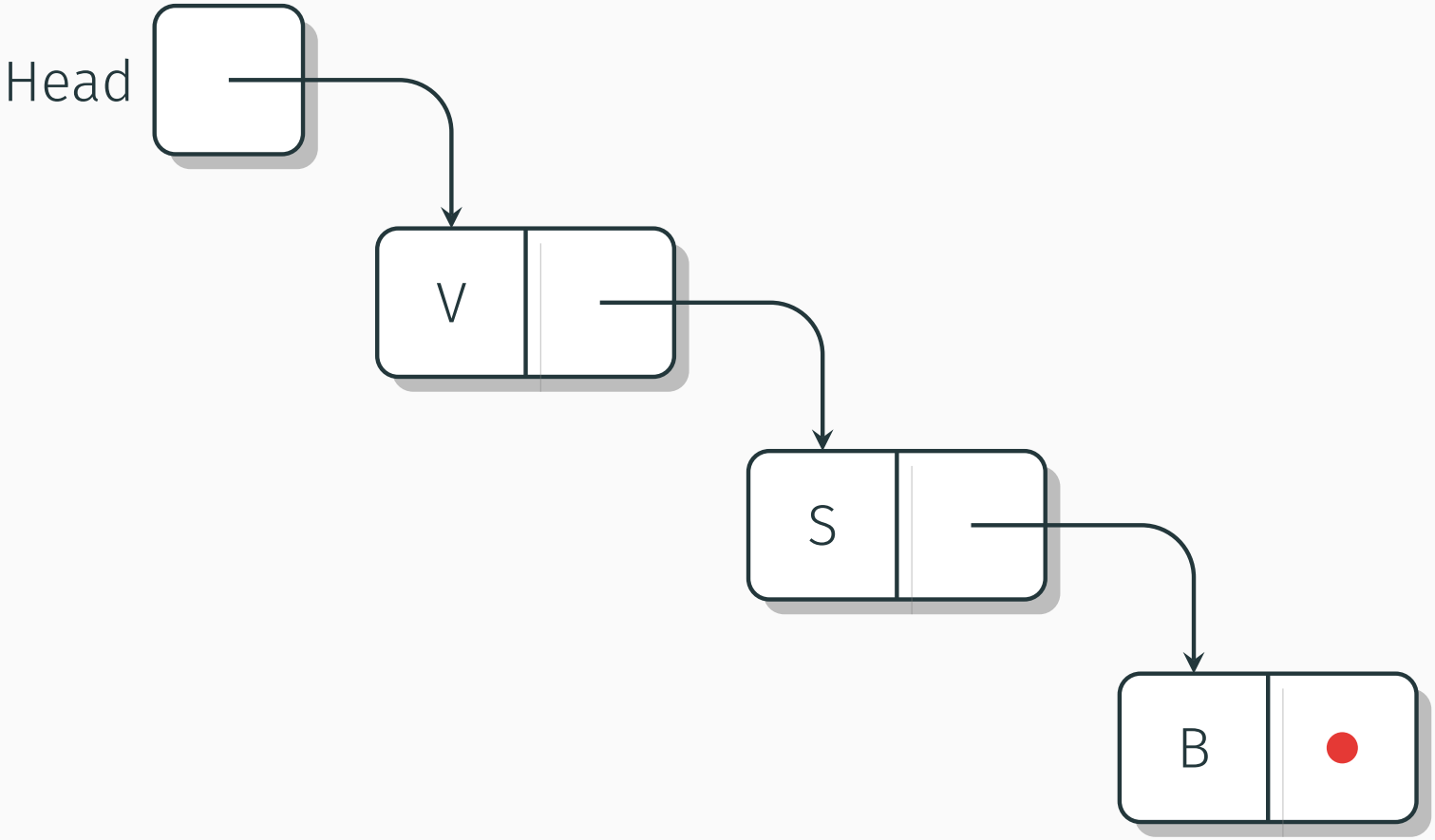
Varianty seznamu

- **jednosměrný seznam** (singly linked list) – nejjednodušší varianta, odkaz pouze na následníka,
- **obousměrný seznam** (doubly linked list) – položka obsahuje odkaz na předchůdce i následníka,
- **kruhový seznam** (circular list) – hlava a ocas seznamu splývají.

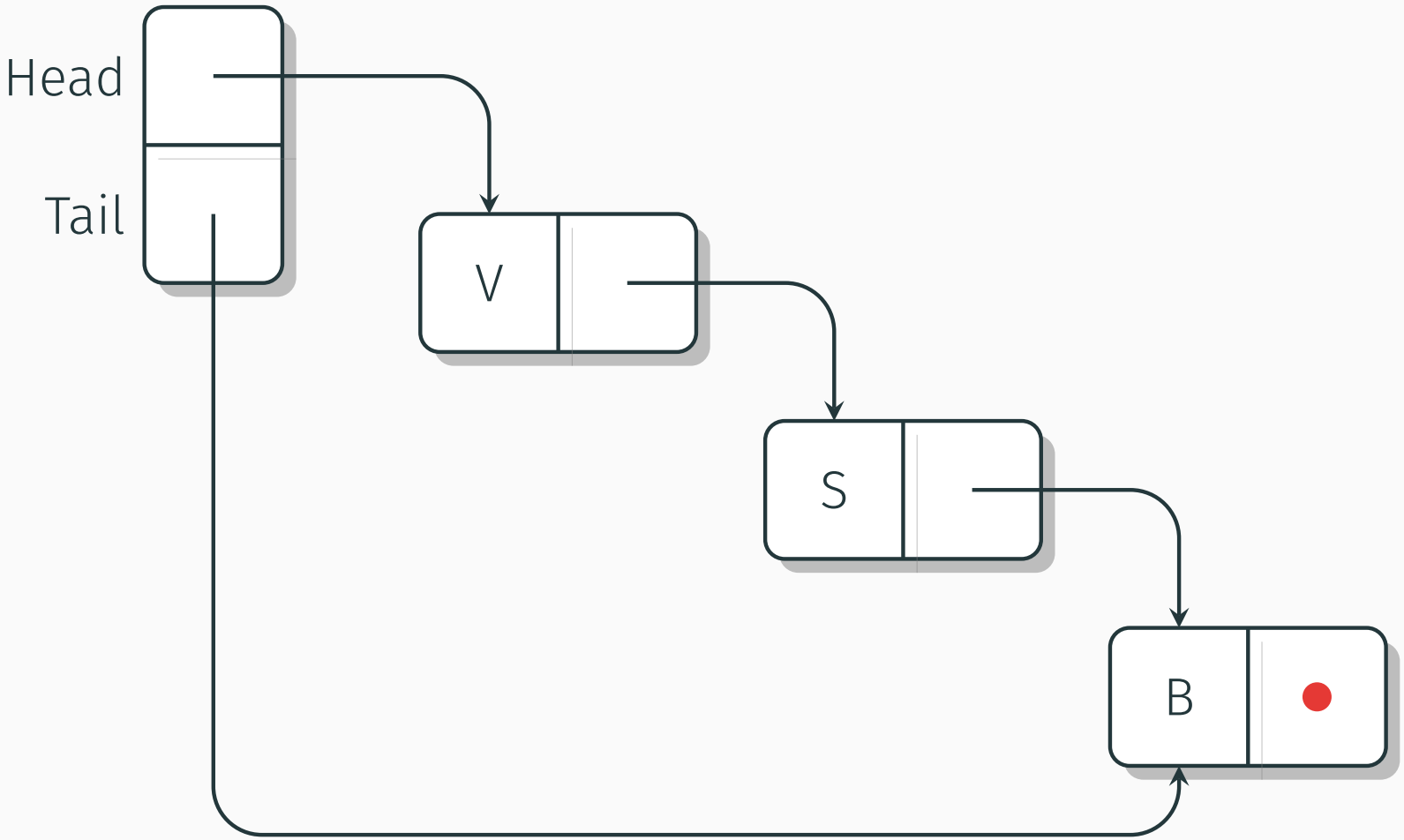
Seznam – jednosměrný seznam

- složen z položek (items),
- položky obsahují data a odkaz na další položku,
- každá položka odkazuje na následovníka,
- k položkám lze přistupovat sekvenčně,
- přímý přístup na základě indexu je složitý (průchod cyklem),
- pohyb seznamem dozadu je také problém,
- konec seznamu – speciální odkaz „nikam“, většinou pojmenován **nil**, **NULL**, **nullptr** či **Nothing**.

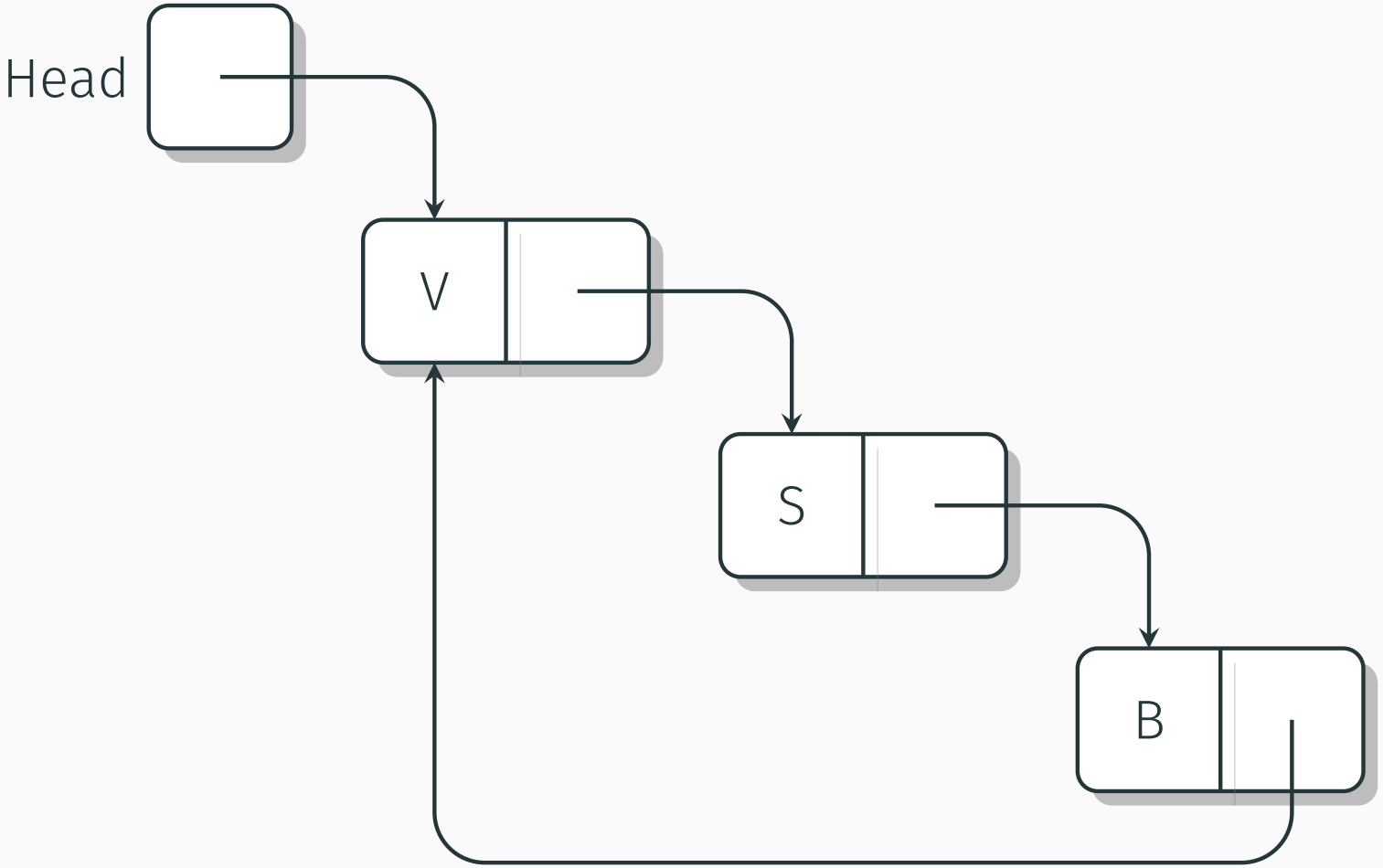
Seznam – jednosměrný, jen hlava seznamu



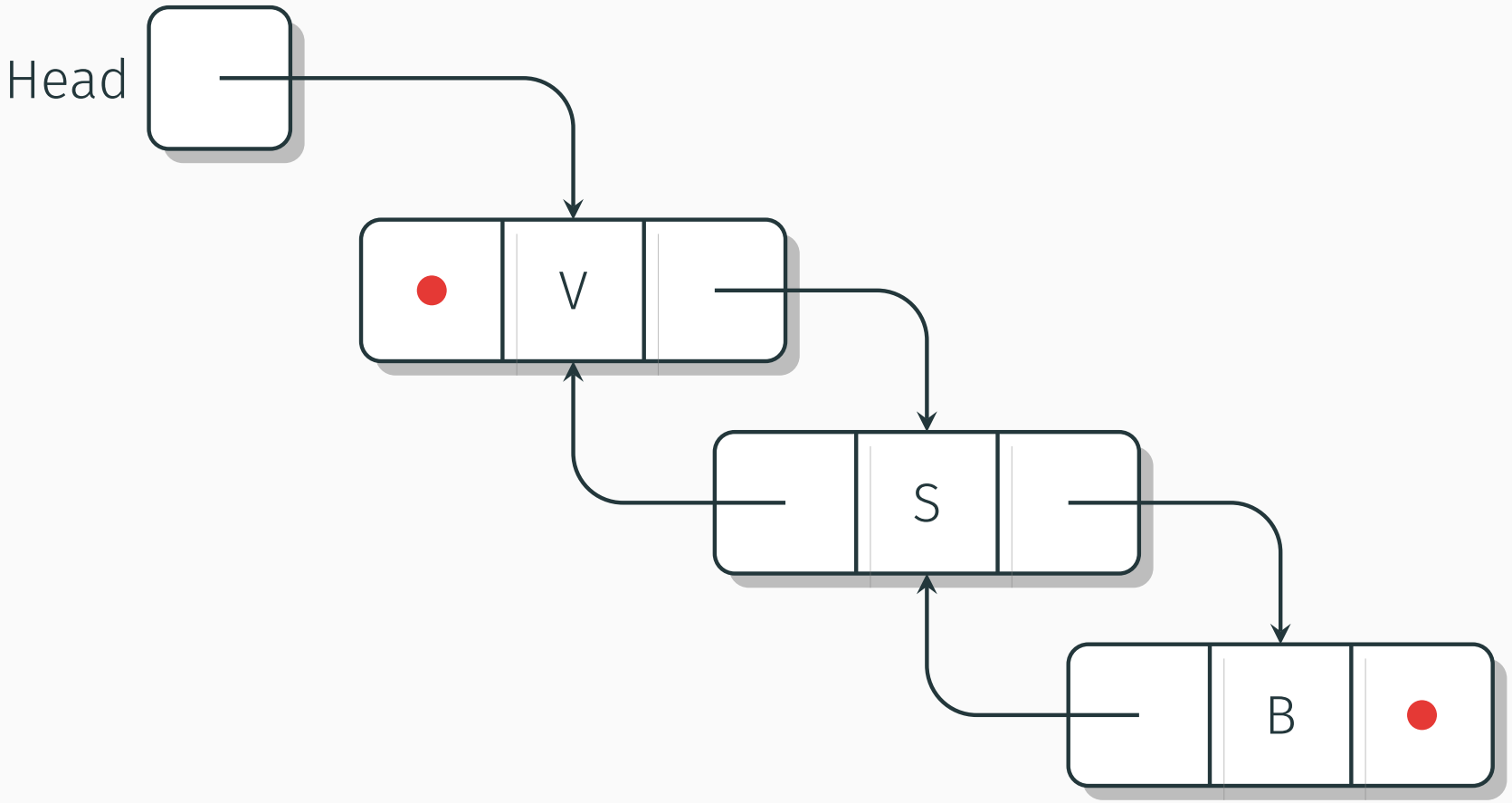
Seznam – jednosměrný, hlava i ocas seznamu



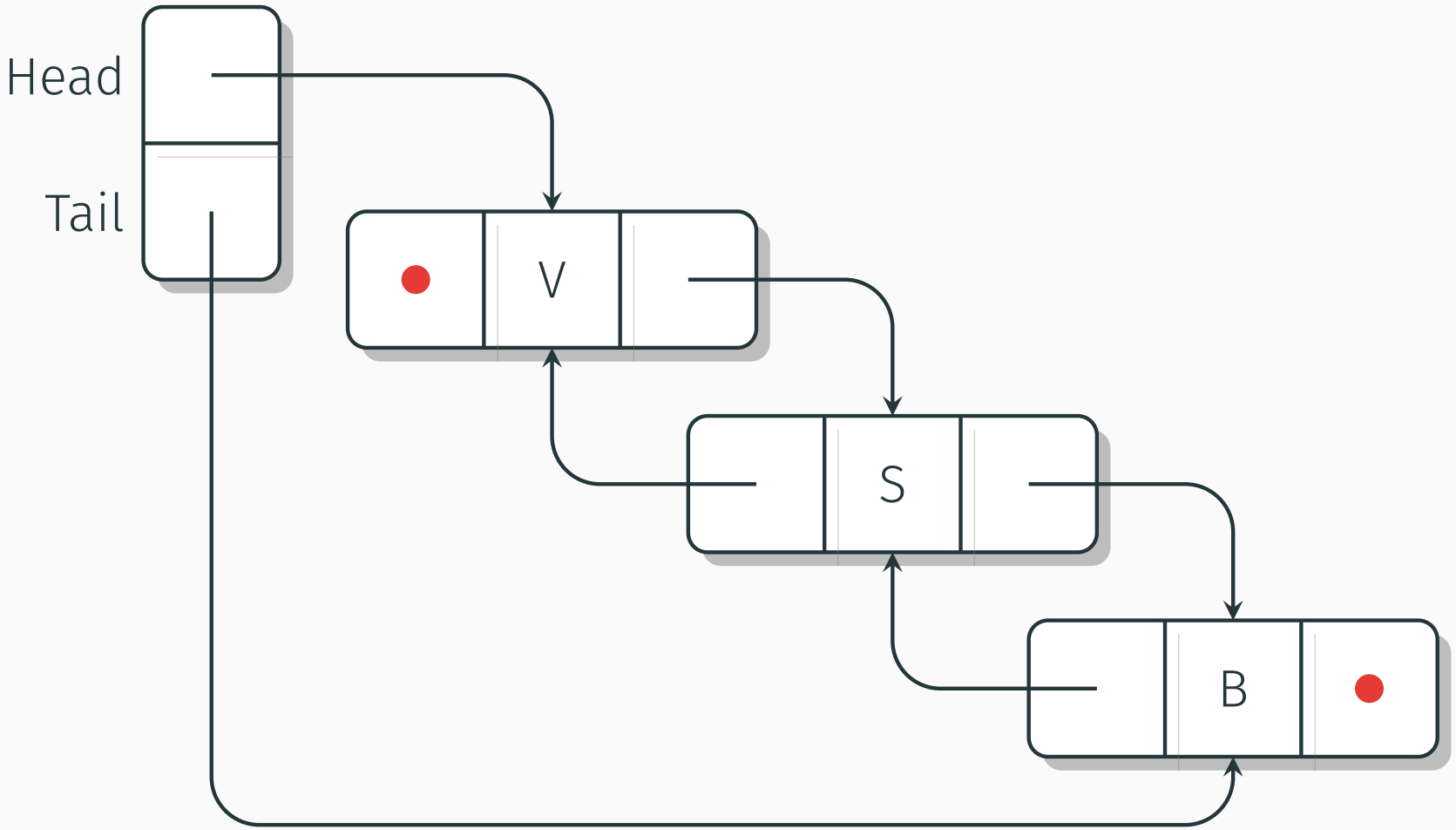
Seznam – jednosměrný, kruhový



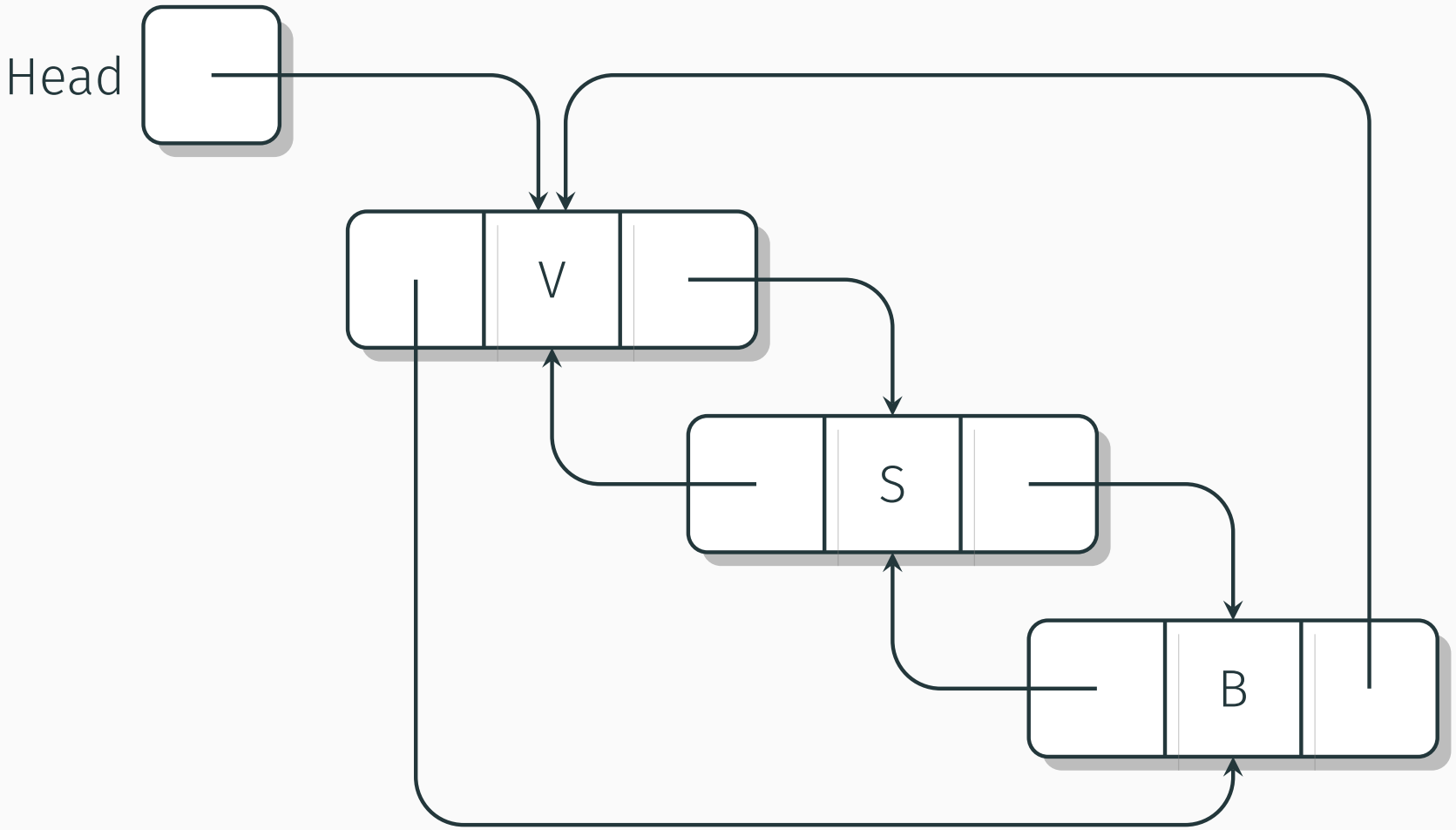
Seznam – obousměrný, jen hlava seznamu



Seznam – obousměrný, hlava i ocas seznamu



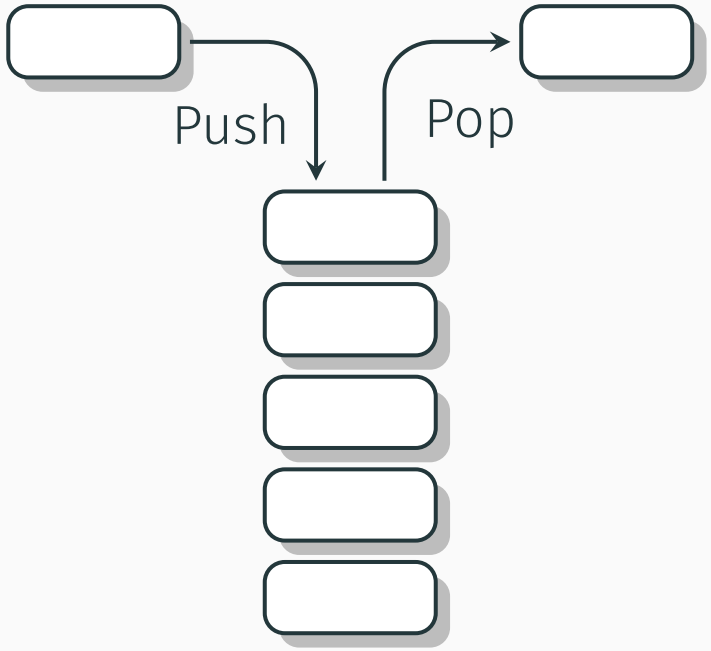
Seznam – obousměrný, kruhový



Zásobník (Stack)

Charakteristika

- princip **last-in, first-out**, **LIFO**
- prvek, který byl vložen poslední, je jako první ze zásobníku vyzvednut



Atributy

- prvky vkládáme na tzv. **vrchol zásobníku** (stack pointer).
- prvně vložený prvek se nazývá **dno zásobníku** (stack bottom)

Zásobník – operace

Základní operace

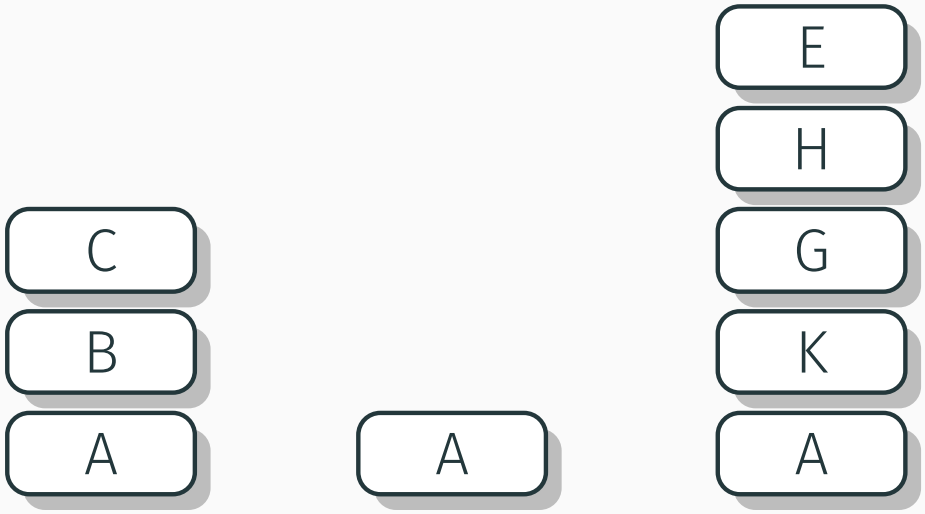
- Push – vložení prvku na vrchol zásobníku
- Pop – vyjmutí prvku z vrcholu zásobníku
- IsEmpty – test prázdnosti zásobníku
- Top – vrátí prvek z vrcholu zásobníku bez jeho vyjmutí

Další možné operace

- Init – inicializace zásobníku
- Clear – vyjmutí všech prvků ze zásobníku
- IsFull – test, zda je zásobník plný (pouze pro zásobník s omezenou kapacitou)

Správně implementované operace mají **konstantní** časovou složitost $O(1)$, tj. jejich časová složitost nezávisí na počtu prvků v zásobníku.

Zásobník – vizualizace




```
Push(A)
Push(B)
Push(C)
```

```
Pop()
Pop()
```

```
Push(K)
Push(G)
Push(H)
Push(E)
```

Zásobník – chybové stavy

- Pokud provedeme operaci Pop na prázdném zásobníku nastává tzv. **podtečení** (stack underflow).
- Pokud není možné přidat další prvek, nastává tzv. **přetečení** (stack overflow).

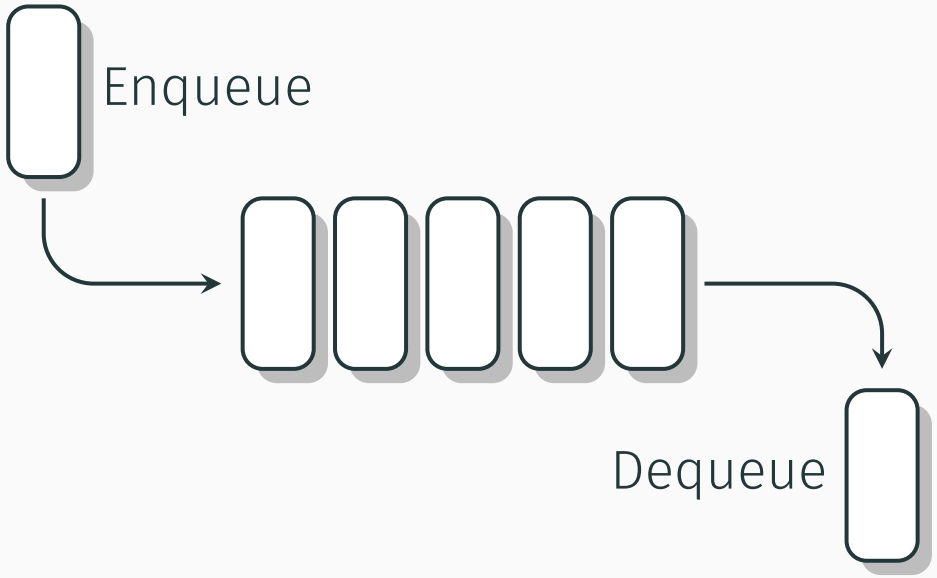
Zásobník – využití zásobníku

- volání funkcí (metod)
- vyhodnocování aritmetických výrazů
- odstranění rekurze
- zásobníkově orientované jazyky, například PostScript, PDF
- testování parity závorek, HTML/XML značek

Fronta (Queue)

Charakteristika

- princip **first-in, first-out**, **FIFO**
- prvek, který byl vložen první, je také jako první z fronty vyzvednut



Atributy

- první prvek se nazývá **hlava** fronty (**head**),
- poslední prvek se nazývá **ocas** fronty (**tail**).

Fronta – operace

Základní operace

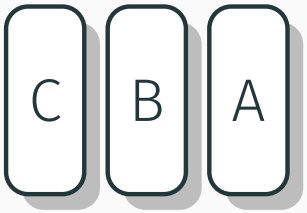
- Enqueue – vložení prvku na konec fronty
- Dequeue – vyjmutí prvku ze začátku fronty
- Peek – vrátí prvek ze začátku fronty bez jeho vyjmutí
- IsEmpty – test zda je fronta prázdná

Další možné operace

- Init – inicializace fronty
- Clear – vyjmutí všech prvků z fronty
- IsFull – test, zda je fronta zaplněna (pouze u fronty s omezenou kapacitou)

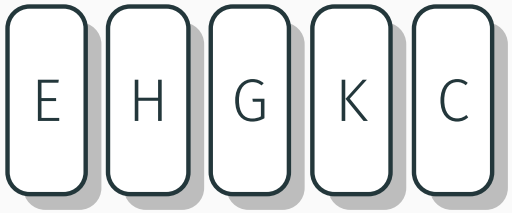
Správně implementované operace mají konstantní časovou složitost $O(1)$, tj. jejich časová složitost nezávisí na počtu prvků ve frontě.

Fronta



Enqueue(A)
Enqueue(B)
Enqueue(C)

Dequeque()
Dequeque()



Enqueue(K)
Enqueue(G)
Enqueue(H)
Enqueue(E)

Fronta – chybové stavy

- Pokud provedeme operaci Dequeue na prázdné frontě nastává tzv. **podtečení** (queue underflow).
- Pokud není možné přidat další prvek, nastává tzv. **přetečení** (queue overflow).

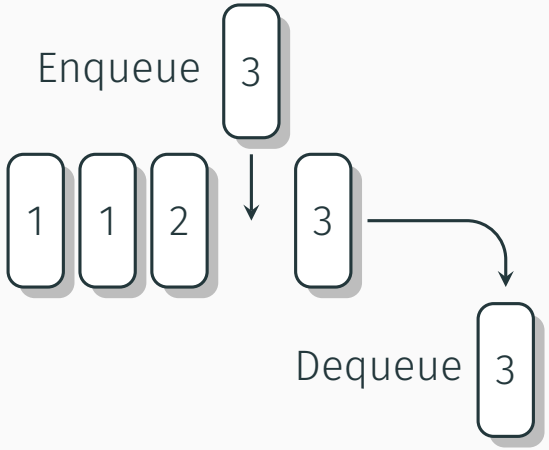
Fronta – využití

- tisková fronta u sdílené tiskárny
- plánovač v operačním systému (více běžících procesů na jednoprocesorovém počítači ⇒ procesy se musí střídat)
- obsluha uživatelů na serverech obecně

Prioritní fronta (Priority Queue)

Charakteristika

- řešení úlohy „Vyjmi z množiny největší prvek a zpracuj ho.“
- na rozdíl od obyčejné fronty se k prvkům váže ještě priorita,
- pro prvky se stejnou prioritou FIFO,
- prvek s vyšší prioritou předbíhá ty s nižší prioritou a odchází z fronty dříve



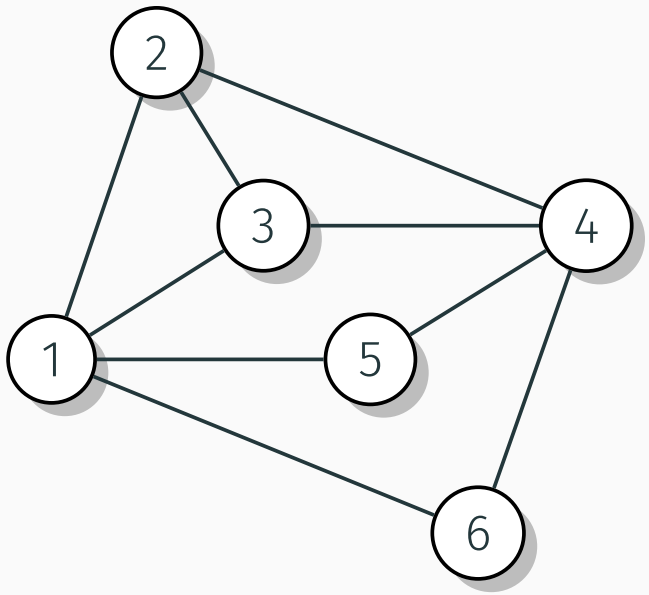
Implementace

- pomocí pole nebo setříděného pole,
- efektivněji pomocí datové struktury zvané **halda** (heap).

Neorientovaný graf

Definice

Neorientovaným grafem nazýváme dvojici $G = (V, E)$, kde V je konečná neprázdná množina **vrcholů**, E je množina jednoprvkových nebo dvouprvkových podmnožin V. Prvky množiny E se nazývají **hrany** grafu.



Příklad

$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$

Hrany v neorientovaném grafu

Mějme hranu $e \in E$, kde $e = \{u, v\}$.

- O hraně e říkáme, že **spojuje** vrcholy u a v.
- Vrcholům u a v říkáme **krajní vrcholy** hrany e.
- Dále říkáme, že vrcholy u a v jsou **incidentní** (nebo, že **incidují**) s hranou e. A obdobně, že hrana e je incidentní s vrcholy u a v.
- Protože hrana e spojuje vrcholy u a v říkáme, o nich že jsou to **sousední** (sousedící) vrcholy.

Definice

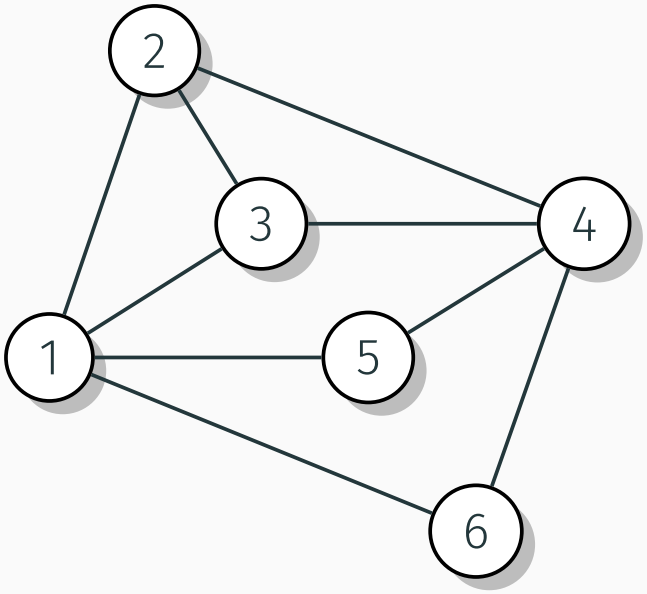
Hranu spojující vrchol se sebou samým nazýváme **smyčkou**.

Neorientovaný graf – stupeň vrcholu

Definice

Stupněm vrcholu v neorientovaném grafu nazýváme počet hran s vrcholem incidentních, tj. $s(v) = |\{e \in E \mid v \in e\}|$.

Příklad



v	$s(v)$
1	4
2	3
3	3
4	4
5	2
6	2

Neorientovaný graf – stupeň vrcholu (pokrač.)

Věta

Součet stupňů vrcholů libovolného neorientovaného grafu $G = (V, E)$ *je roven dvojnásobku počtu jeho hran.*

$$\sum_{v \in V} s(v) = 2|E|$$

Důkaz.

Zřejmý (v sumě se každá hrana počítá dvakrát). □

Počet hran v neorientovaném grafu

Věta

Pro libovolný neorientovaný graf $G = (V, E)$ *bez smyček platí:*

$$0 \leq |E| \leq \frac{1}{2}|V|(|V| - 1)$$

Důkaz.

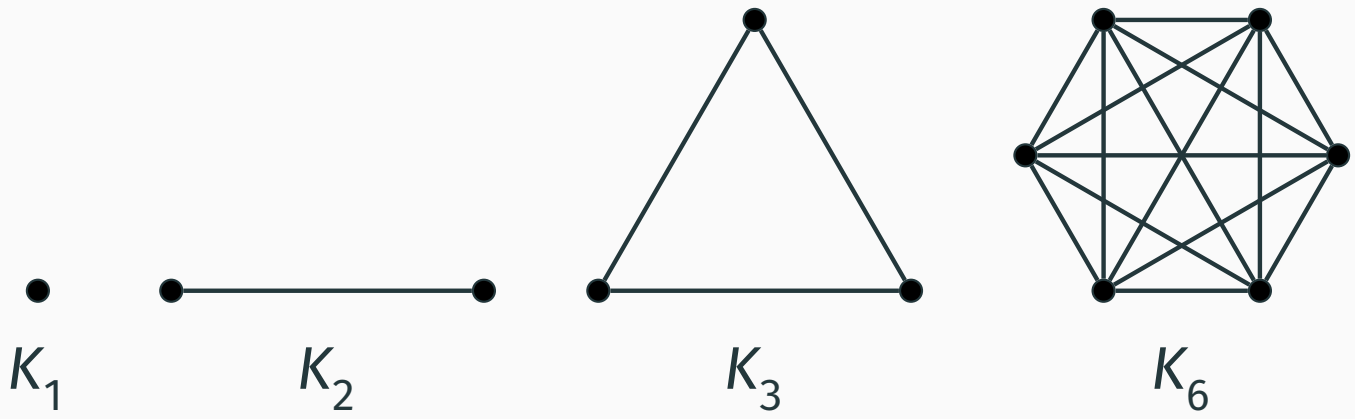
Maxmálního počtu hran v grafu docílíme tak, že každý z $|V|$ vrcholů spojíme hranou se všemi ostatními vrcholy, kterých je $|V| - 1$. Součin $|V|(|V| - 1)$ musíme vydělit dvěma, protože jsme každou hranu započítali dvakrát. □

Úplný graf

Definice

Neorientovaný graf $G = (V, E)$ ve kterém pro každou dvojici vrcholů u a v existuje hrana nazýváme **úplným grafem** a označujeme je $K_{|V|}$.

Příklad



Hustý vs. řídký graf

- **Hustý graf** – „téměř“ kompletní graf, chybí jen „relativně“ malý počet hran do maximálního počtu
- **Řídký graf** – „velice malý“ počet hran, „relativně“ velký počet hran neexistuje.
- Přesná definice není, pojmy jako „téměř“, „relativně“ či „velice malý“ jsou subjektivní.
- Záleží vždy na konkrétní situaci.
- Při výběru reprezentace grafu v počítači je nutno brát zřetel na to zda je graf hustý nebo řídký. A s tím následně souvisí časová složitost implementovaných algoritmů.

Podgraf

Definice

Graf $H = (V_H, E_H)$ nazýváme podgrafem grafu $G = (V_G, E_G)$, jestliže platí následující podmínky:

1. $V_H \subseteq V_G$
2. $E_H \subseteq E_G$
3. Hrany grafu H mají oba vrcholy v H.

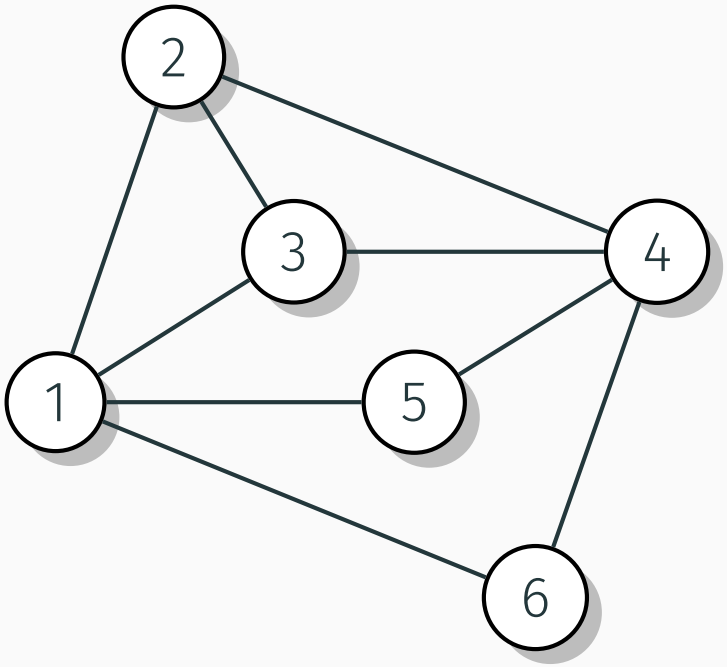
Podgraf (pokrač.)

Poznámky

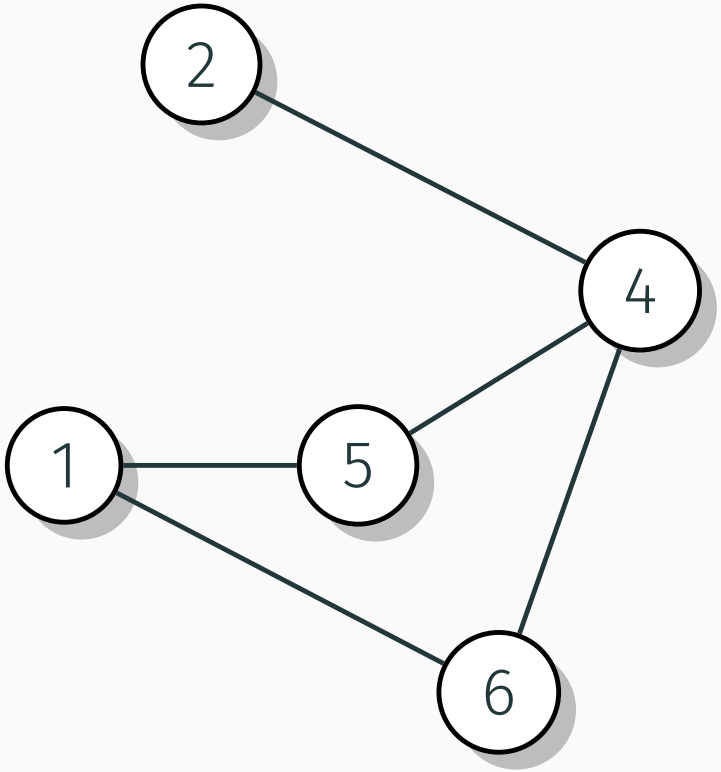
- Jinými slovy, podgraf vznikne vymazáním některých vrcholů původního grafu, všech hran do těchto vrcholů zasahujících a případně některých dalších hran.
- Termín podgraf se v teorii grafů používá jako jistá obdoba pojmu podmnožina.

Podgraf (pokrač.)

Příklad



Graf G

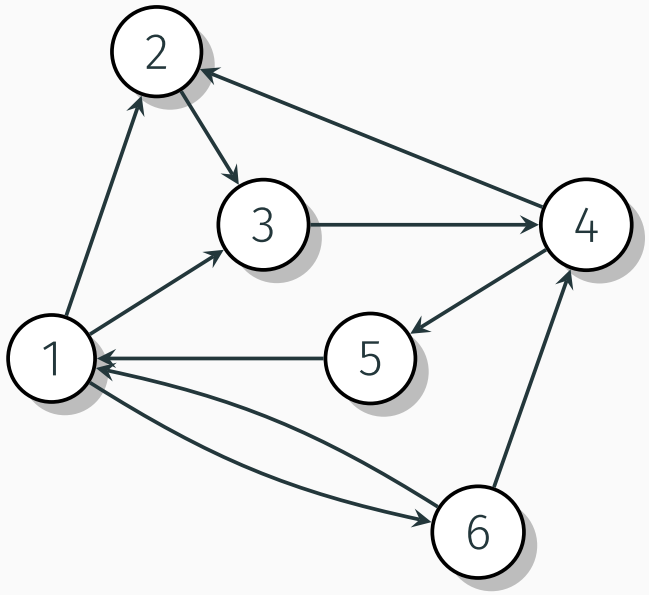


Podgraf H

Orientovaný graf

Definice

Orientovaným grafem nazýváme dvojici $G = (V, E)$, kde V je konečná neprázdná množina **vrcholů**, E je množina uspořádaných dvojic (u, v), **hran**, z kartézského součinu $V \times V$, neboli $(u, v) \in V \times V$.



Příklad

$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

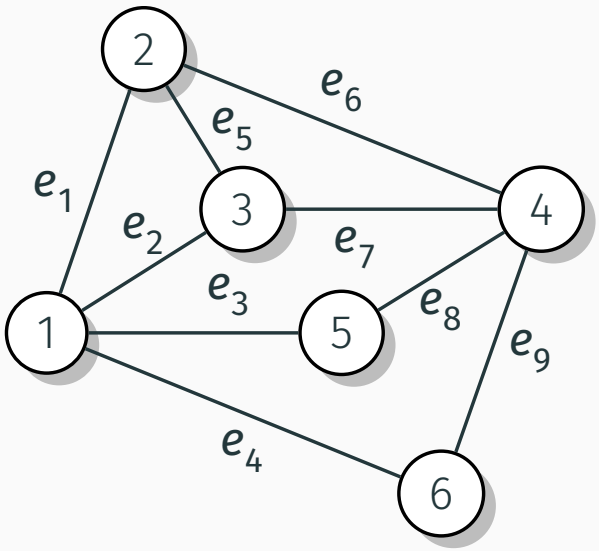
$E = \{(1, 2), (1, 3), (1, 6), (2, 3), (3, 4), (4, 2), (4, 5), (5, 1),$
$\{(6, 1), (6, 4)\}$

Způsoby reprezentace grafu

- grafickou formou:
  - prostě obrázkem,
  - asi nejsrozumitelnější forma pro člověka,
  - vhodné pro grafy s malým počtem vrcholů,
  - prakticky nemožnost zpracování počítačem.
- maticí,
- seznamem sousedících vrcholů.

Matice incidence

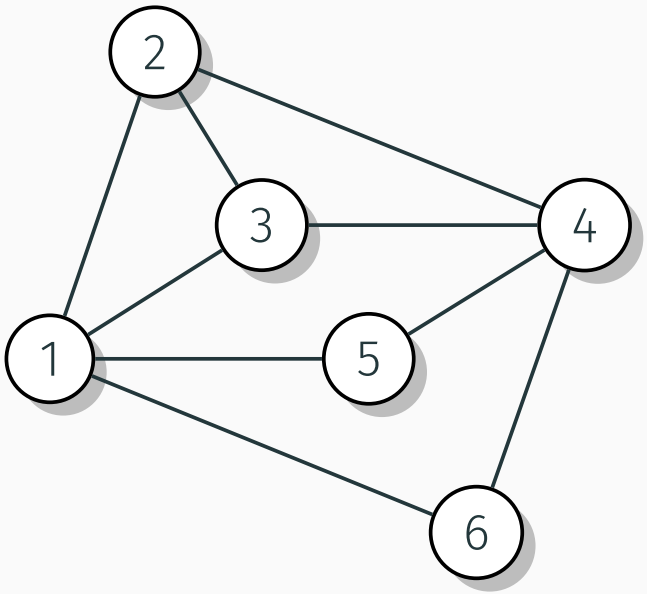
- Počet řádků matice odpovídá počtu vrcholů, počet sloupců odpovídá počtu hran.
- Pokud je vrchol incidentní s hranou, je na dané pozici jednička, jinak nula.



$$
\begin{array}{c}
 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6
\end{array}
\begin{array}{c}
\begin{array}{ccccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \end{array} \\
\begin{pmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\end{array}
$$

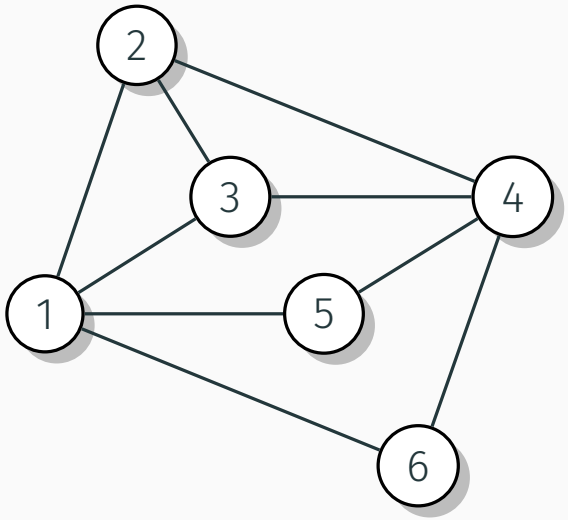
Matice sousednosti

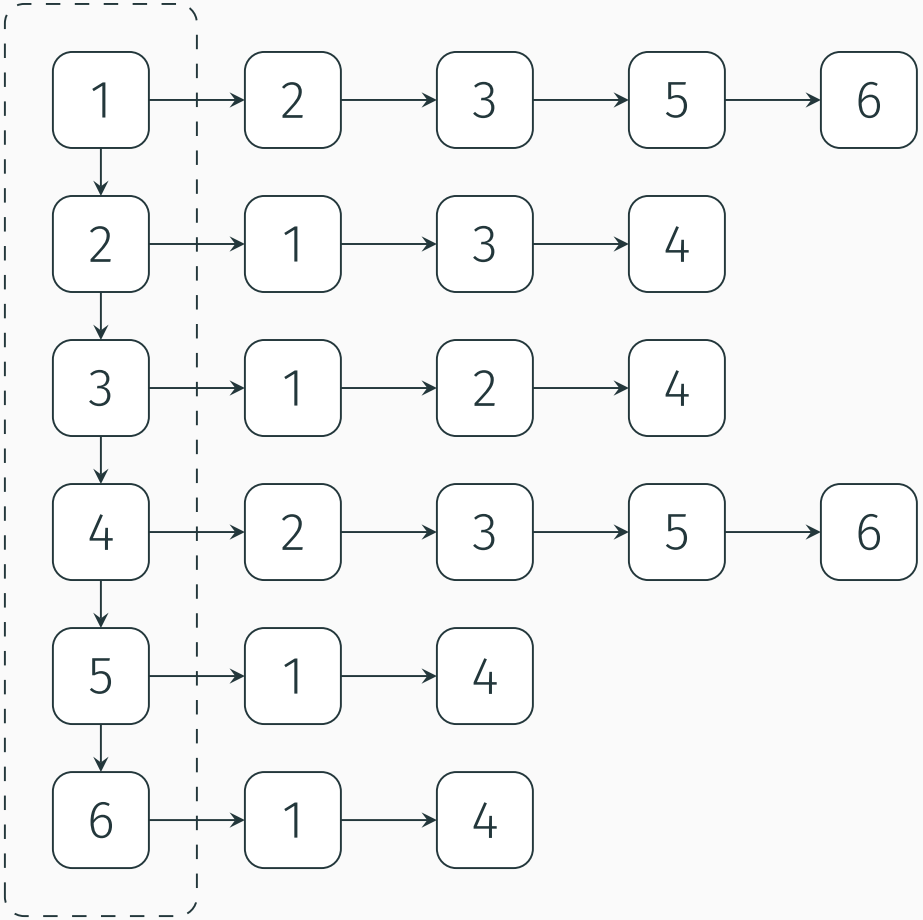
- Čtvercová matice, kde řád matice odpovídá počtu vrcholů v grafu.
- Pokud jsou dva vrcholy spojeny hranou, je na dané pozici jednička, jinak nula.



$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 1 & 0 & 1 & 1 \\
2 & 1 & 0 & 1 & 1 & 0 & 0 \\
3 & 1 & 1 & 0 & 1 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 & 1 \\
5 & 1 & 0 & 0 & 1 & 0 & 0 \\
6 & 1 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

Seznam sousedících vrcholů





Hustý vs. řídký graf a jeho reprezentace

Seznam sousedících vrcholů

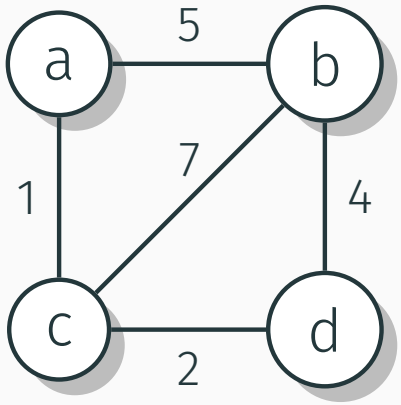
- ukazatele v seznamech zabírají paměť navíc,
- vhodný pro řídké grafy,
- pohodlnější změny struktury grafu (vložení smazání vrcholu, stejně tak hrany).

Maticová reprezentace

- vhodná pro husté grafy,
- vložení či smazání vrcholu je komplikované.

Vážené grafy

- Každé hraně je přiřazeno číslo označované jako **váha** či **cena** hrany.
- Motivace v reálném světě – délka cesty, kapacita datové linky atd.
- Vážené grafy mohou být orientované i neorientované.
- Reprezentace:
  - matice sousednosti – hodnota v matici udává váhu hrany nebo je tu speciální hodnota pro neexistující hranu, ∞
  - seznam sousedících vrcholů – v seznamu sousedů uložíme i váhu konkrétní hrany.

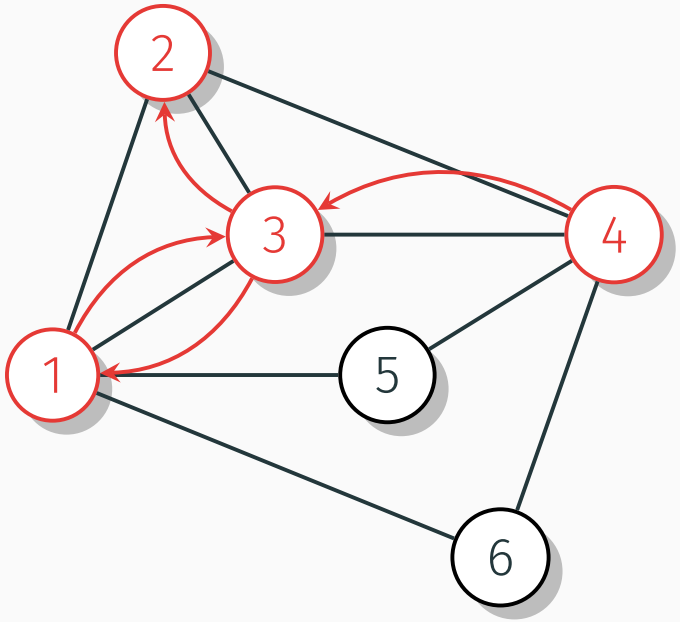


$$\begin{pmatrix} \infty & 5 & 1 & \infty \\ 5 & \infty & 7 & 4 \\ 1 & 7 & \infty & 2 \\ \infty & 4 & 2 & \infty \end{pmatrix}$$

Sled

Definice

Posloupnost navazujících vrcholů a hran
$v_1, e_1, v_2, \dots, v_n, e_n, v_{n+1}$, kde $e_i = \{v_i, v_{i+1}\}$ pro $1 \leq i \leq n$
nazýváme (neorientovaným) **sledem**.



Sled

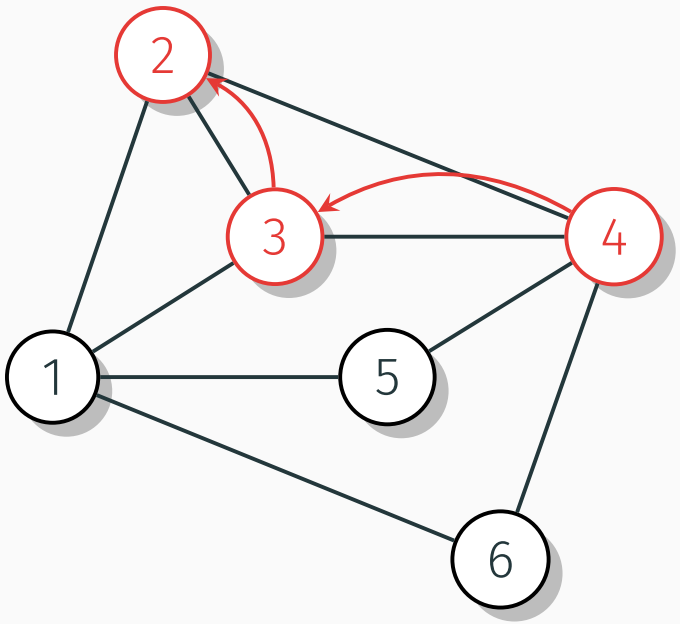
$4\ \{4,3\}\ 3\ \{3,1\}\ 1\ \{1,3\}\ 3\ \{3,2\}\ 2$

V orientovaném grafu mluvíme o orientovaném sledu.

Cesta

Definice

Sled, v němž se neopakuje žádný vrchol nazýváme **cestou**. Tedy $v_i \neq v_j, \forall\ 1 \leq i \leq j \leq n$. Číslo n pak nazýváme **délkou cesty**.



Cesta

4 3 2

Z faktu, že se v cestě neopakují vrcholy, vyplývá, že se v ní neopakují ani hrany. Každá cesta je tedy zároveň i sledem.

V orientovaném grafu mluvíme o orientované cestě.

Souvislost grafu

Definice

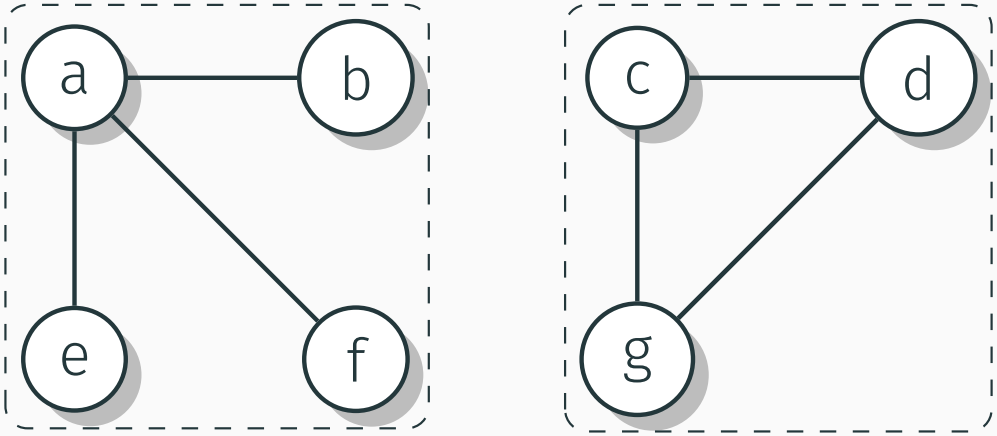
Graf se nazývá **souvislý**, jestliže mezi každými dvěma vrcholy existuje cesta.

Nesouvislý graf se skládá z několika souvislých částí tzv. souvislých komponent.

Definice

Souvislá komponenta grafu je maximální souvislý podgraf daného grafu.

Souvislost grafu (pokrač.)



Věta

Nechť $G = (V, E)$ *je souvislý graf. Pak platí* $|E| \geq |V| - 1$.

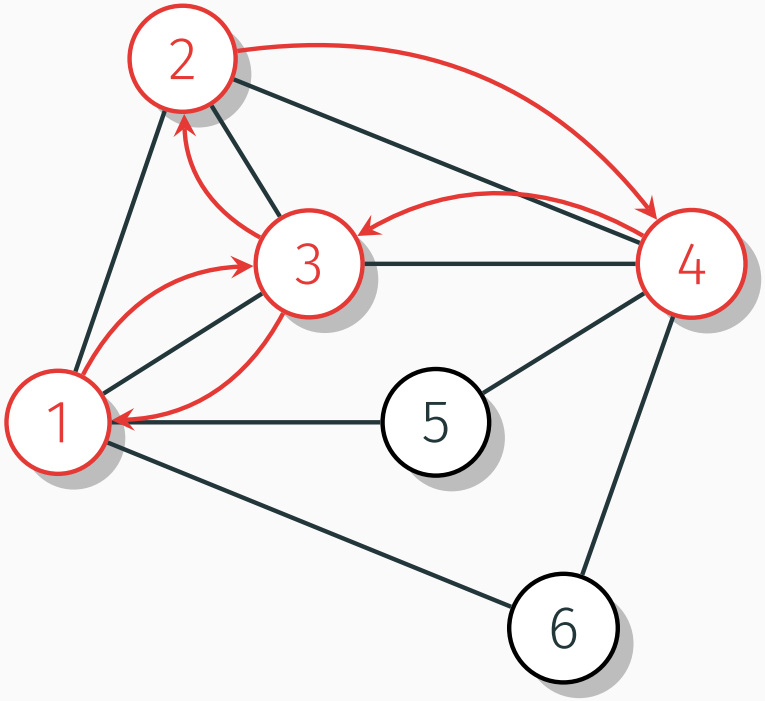
Důkaz.

Zřejmý. □

Uzavřený sled

Definice

Sled, který má alespoň jednu hranu a jehož počáteční a koncový vrchol splývají, nazýváme **uzavřeným sledem**.



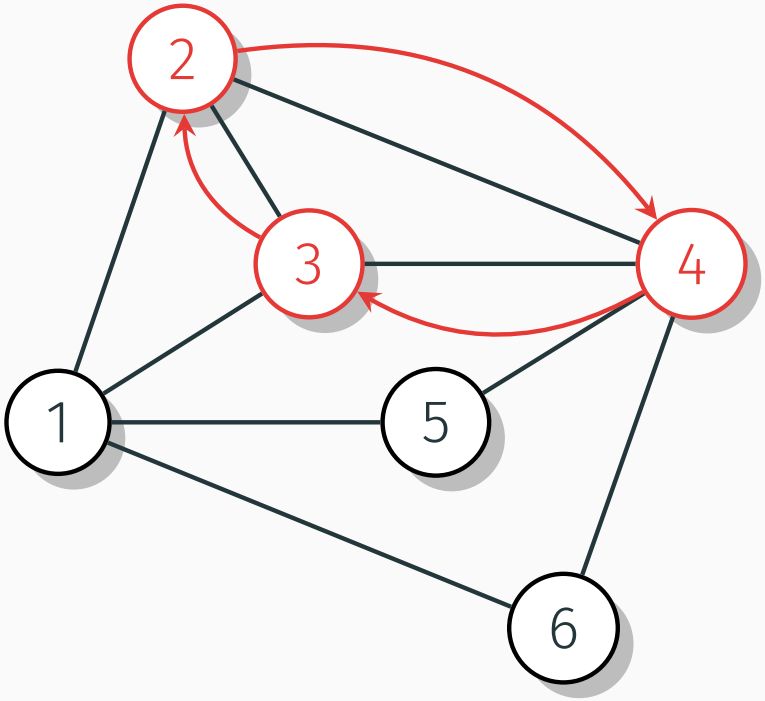
Uzavřený sled

$4\ \{4,3\}\ 3\ \{3,1\}\ 1\ \{1,3\}\ 3\ \{3,2\}$
$2\ \{2,4\}\ 4$

Kružnice

Definice

Uzavřená cesta je uzavřený sled, v němž se neopakují vrcholy ani hrany. Uzavřená cesta se nazývá také **kružnice**.



Kružnice

4 3 2

V definici kružnice jsme museli zakázat kromě opakování vrcholů i opakování hran proto, aby posloupnost v_1, e_1, v_2, e_1, v_1 nemohla být považována za kružnici.

Acykličnost grafu

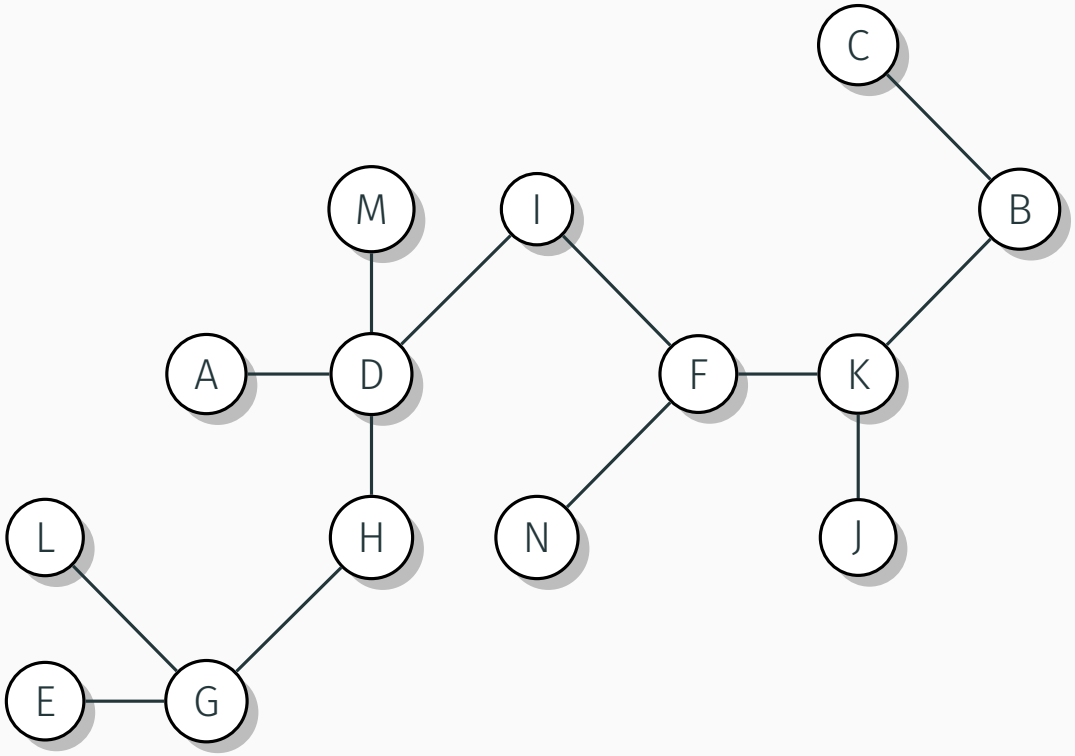
Definice

Graf se nazývá **acyklický**, jestliže neobsahuje kružnici.

Volný strom

Definice

Souvislý, acyklický, neorientovaný graf nazýváme **volným stromem** (angl. free tree).



Poznámka

Prázdný graf je možné považovat za strom, tzv. prázdný strom.

Volný strom (pokrač.)

Terminologie

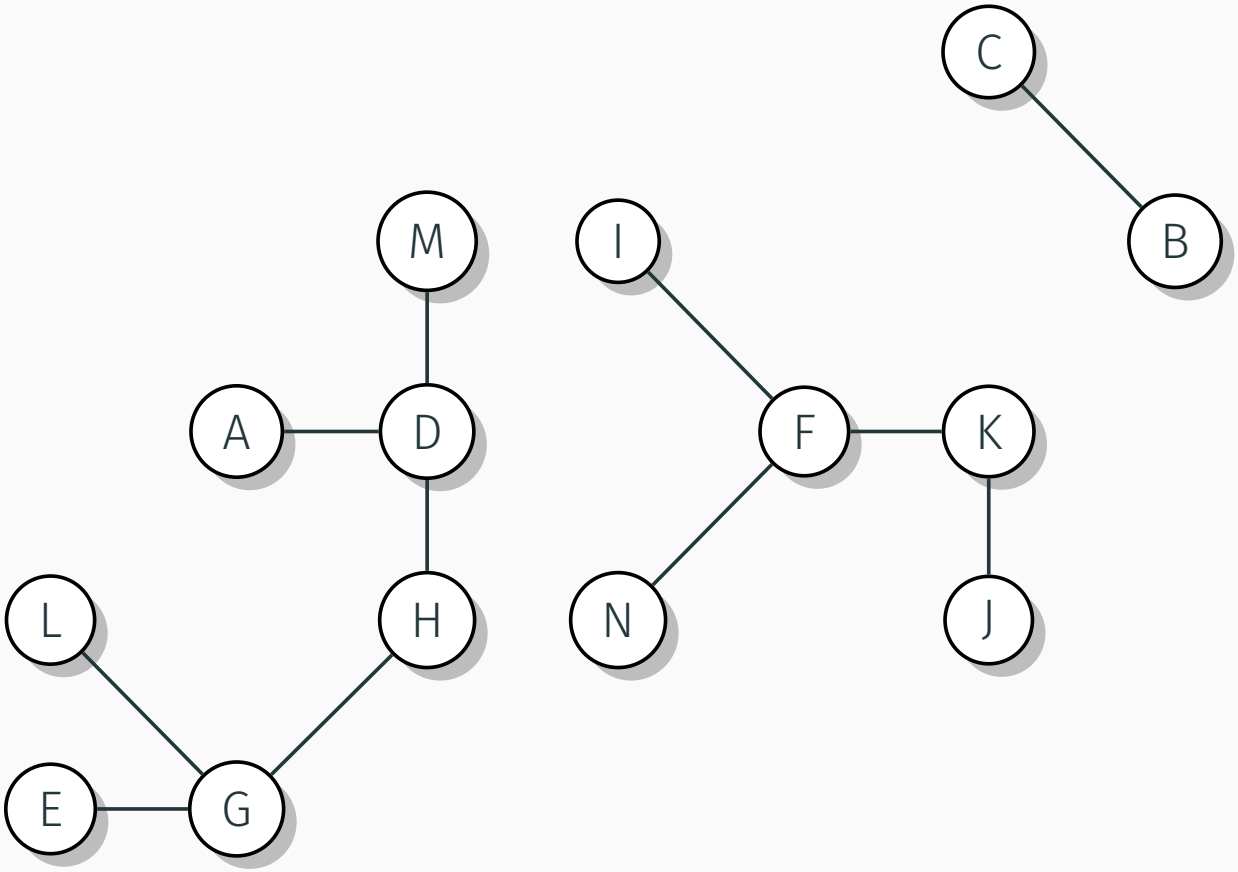
- V teorii grafů se objekty, které propojujeme hranami nazývají obvykle vrcholy (angl. vertex, vertices).
- Pokud mluvíme o stromech lze pro vrchol používat i výraz **uzel** (angl. node).
- Označení vrchol a uzel je rovnocenné, jde spíše o zvyklost.

Les

Definice

Acyklický graf, který není spojitý se nazývá **les** (angl. forest).

Každá souvislá komponenta lesa je volný strom



Vlastnosti volného stromu

Věta

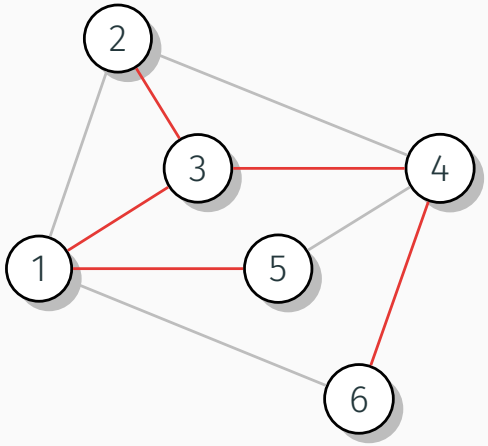
Nechť $G = (V, E)$ *je neorientovaný graf, potom následující tvrzení jsou ekvivalentní:*

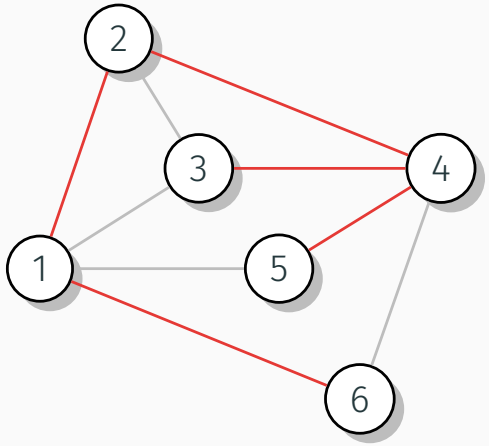
1. G *je volný strom.*
2. *Každé dva vrcholy v* G *jsou spojeny právě jednou cestou.*
3. G *je souvislý, ale pokud odebereme libovolnou hranu, získáme nesouvislý graf.*
4. G *je souvislý, a* $|E| = |V| - 1$.
5. G *je acyklický, a* $|E| = |V| - 1$.
6. G *je acyklický. Přidáním jediné hrany do množiny hran* E *bude výsledný graf obsahovat kružnici.*

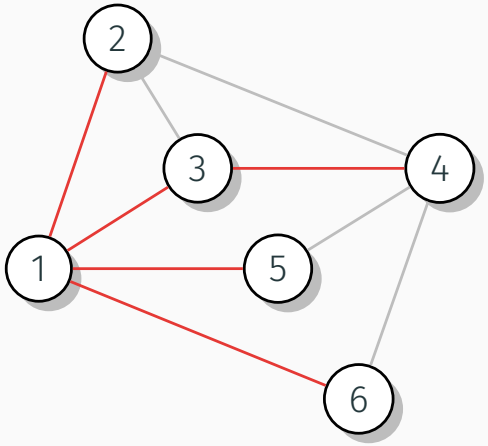
Kostra grafu (angl. Spanning tree)

Definice

Kostrou souvislého grafu G nazýváme takový podgraf grafu G na množině všech jeho vrcholů, který je stromem.







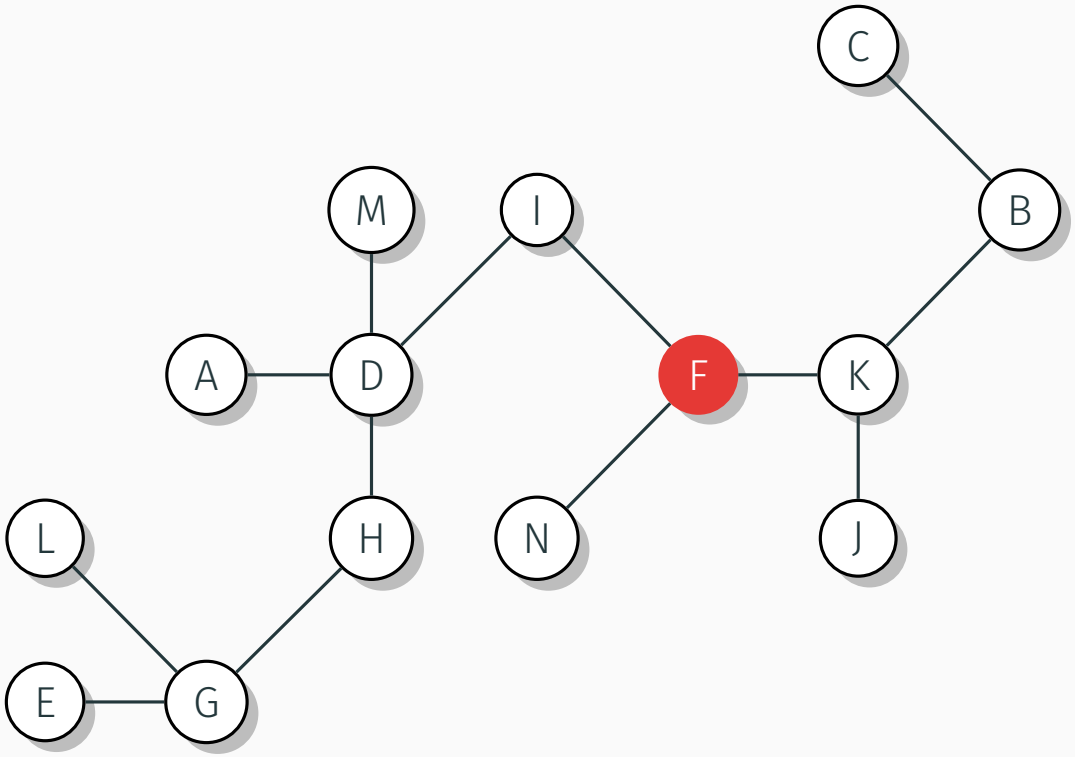
Poznámky

- Kostra musí obsahovat všechny vrcholy původního grafu G.
- Koster grafu může být více.

Kořenový strom

Definice

Volný strom, který obsahuje jeden odlišný vrchol, se nazývá **kořenový strom** (angl. rooted tree). Odlišný vrchol se nazývá **kořen** stromu.

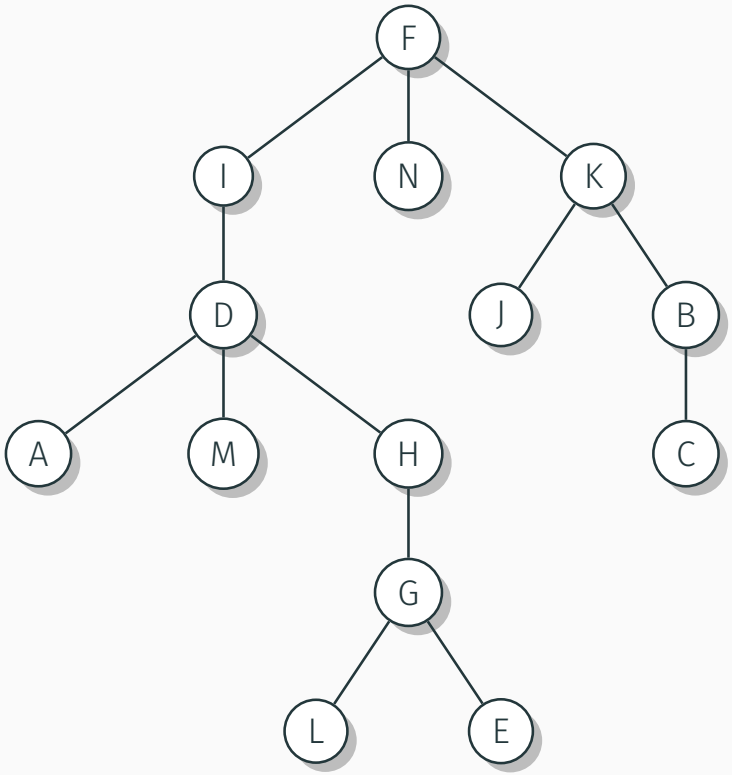


Poznámka

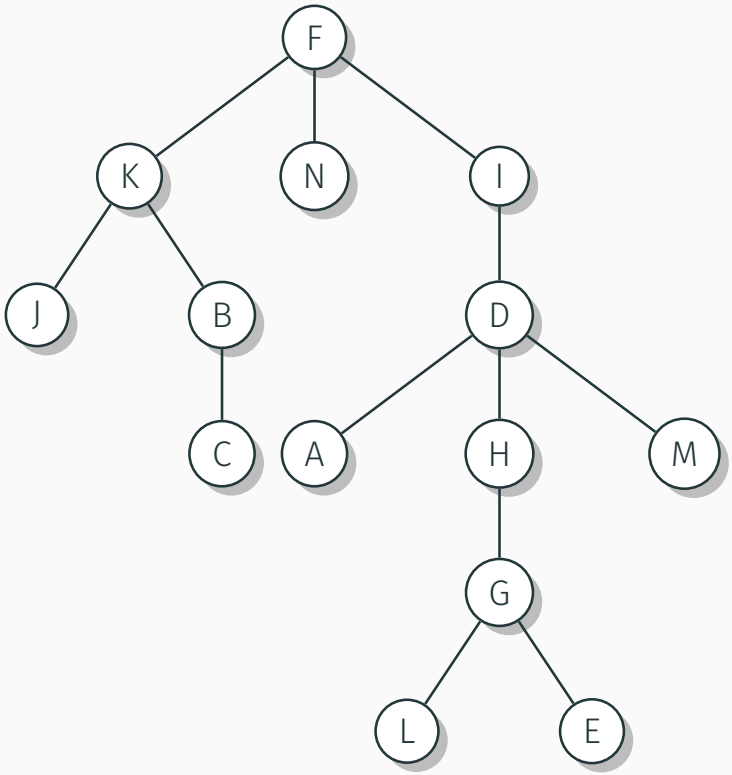
Někdy se používá také označení **zakořeněný strom**.

Kořenový strom – obvyklá vizualizace

Vizualizace 1



Vizualizace 2



Obě vizualizace jsou kořenový strom **rovnocenné**! Neexistuje „vlevo“ či „vpravo“.

Kořenový strom – základní pojmy

Uvažujme vrchol x v kořenovém stromu T s kořenem r.

- Libovolný vrchol y na jednoznačné cestě od kořene r do vrcholu x se nazývá **předchůdce** vrcholu x.
- Jestliže y je předchůdce x, potom x se nazývá **následovník** vrcholu y.
- Jestliže poslední hrana na cestě z kořene r do vrcholu x je hrana (y, x), potom se vrchol y nazývá **rodič** vrcholu x a vrchol x je **potomek** vrcholu y.
- Dva vrcholy mající stejného rodiče se nazývají **sourozenci**.
- Vrchol bez potomků se nazývá vnější vrchol nebo-li **list**.
- Nelistový vrchol se nazývá **vnitřním** vrcholem stromu.

Kořenový strom – základní pojmy (pokrač.)

Poznámky

- Každý vrchol je pochopitelně předchůdcem a následovníkem sama sebe.
- Jestliže y je předchůdce x a zároveň $x \neq y$, potom y je vlastní předchůdce vrcholu x a x je vlastní následovník vrcholu y.
- Kořen stromu je jediným vrcholem ve stromu bez rodiče.
- Vrchol je obecný pojem. Každý list a vnitřní vrchol je zároveň vrchol (bez přívlastku). Srovnej: člověk, žena, muž.

Stupeň vrcholu

Definice

Počet potomků vrcholu x v kořenovém stromu se nazývá **stupeň vrcholu** x.

Poznámky

- Metoda výpočtu stupně vrcholu se u kořenového stromu liší od výpočtu ve volném stromu.
- V kořenovém stromu se nepočítá rodič.
- Ve volném stromu pojem rodiče neexistuje, existují jen sousední vrcholy, počítají se tudíž všechny vrcholy.

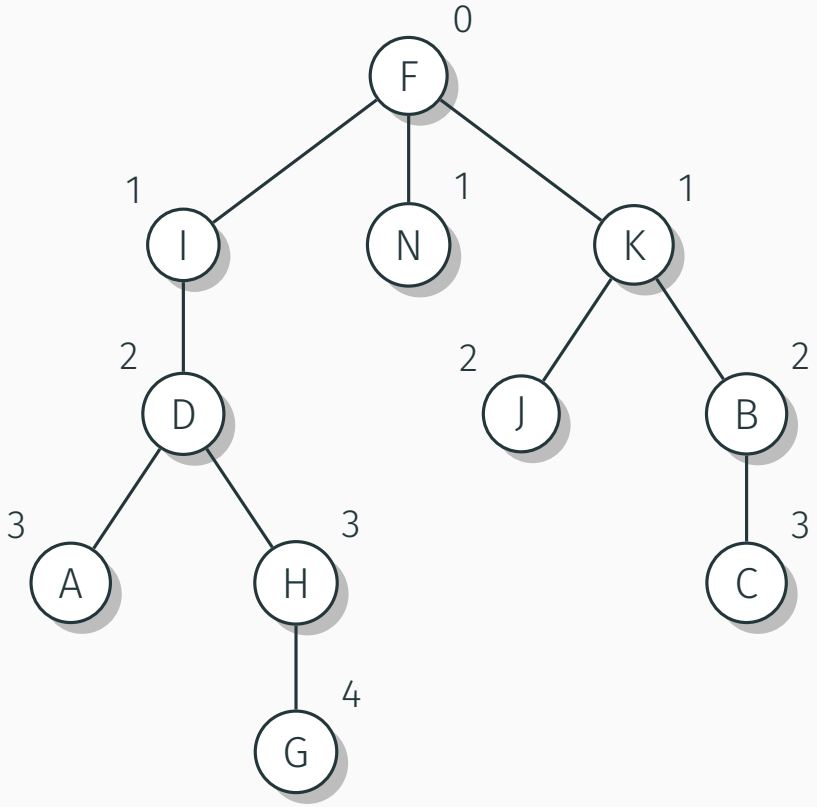
Hloubka vrcholu, výška stromu

Definice

Délka cesty od kořene stromu k vrcholu x se nazývá **hloubka vrcholu** x ve stromu T.

Definice

Největší hloubka libovolného vrcholu se nazývá **výška stromu** T.



Výška stromu je 4.

Seřazený strom

Definice

Kořenový strom ve kterém je určeno pořadí potomků se nazývá **seřazený strom** (angl. ordered tree).

Poznámky

- Tudíž, pokud vrchol má k potomků, lze určit prvního potomka, druhého potomka, až k-tého potomka.
- Pokud ale například prvního potomka zrušíme, ostatní potomci se posouvají! Druhý potomek se stane prvním, druhý třetím atd. Nelze mít mezi potomky „prázdnou pozici“.

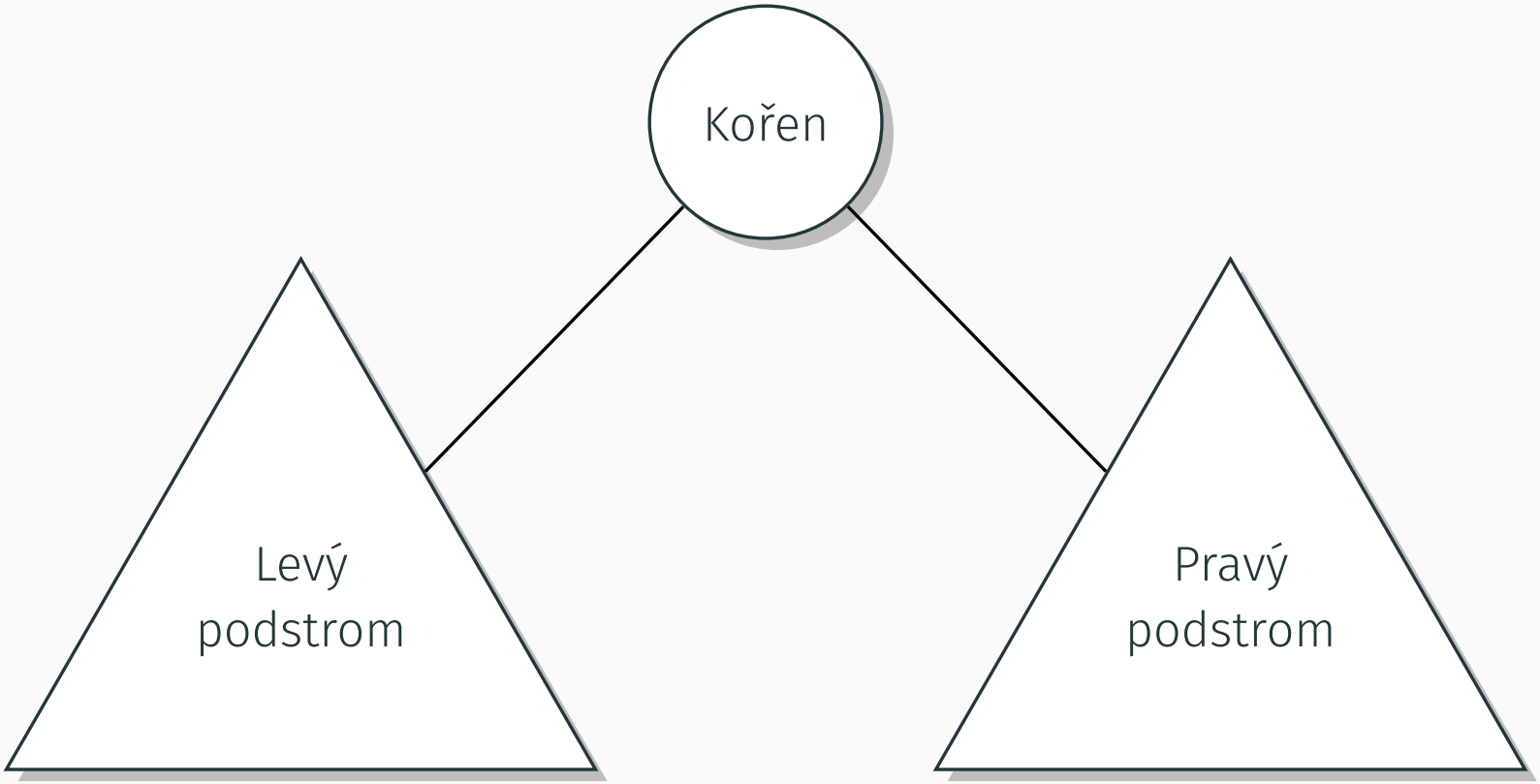
Binární strom

Definice

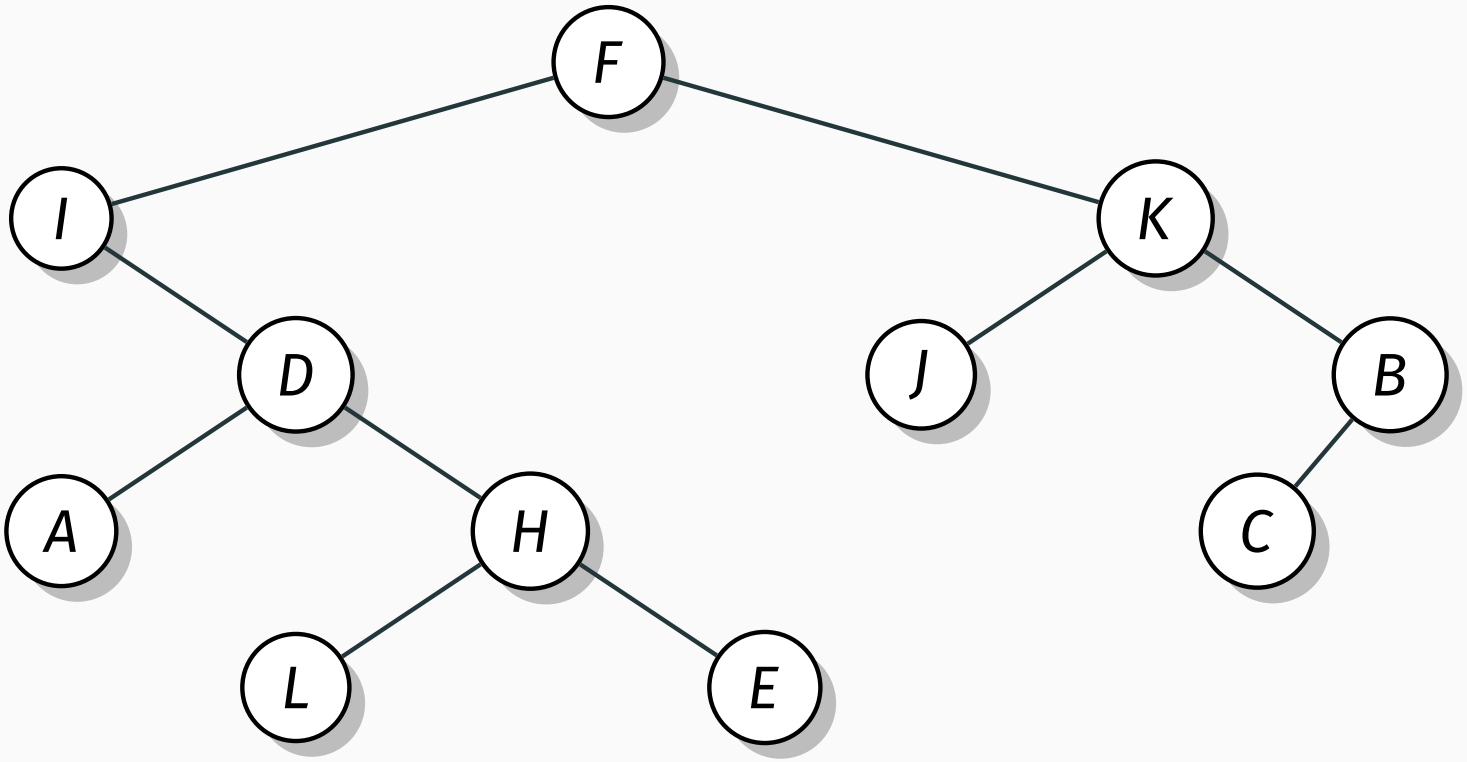
Binární strom je struktura definovaná nad konečnou množinou uzlů M, která:

- **Pravidlo 1**
  neobsahuje žádný uzel, tj. $M = \emptyset$ nebo
- **Pravidlo 2**
  je složena ze tří disjunktních množin uzlů L, R a $\{r\}$, $L \cup R \cup \{r\} = M$:
  - kořene stromu r,
  - binárního stromu nad množinou L zvaného levý podstrom a
  - binárního stromu nad množinou R zvaného pravý podstrom.

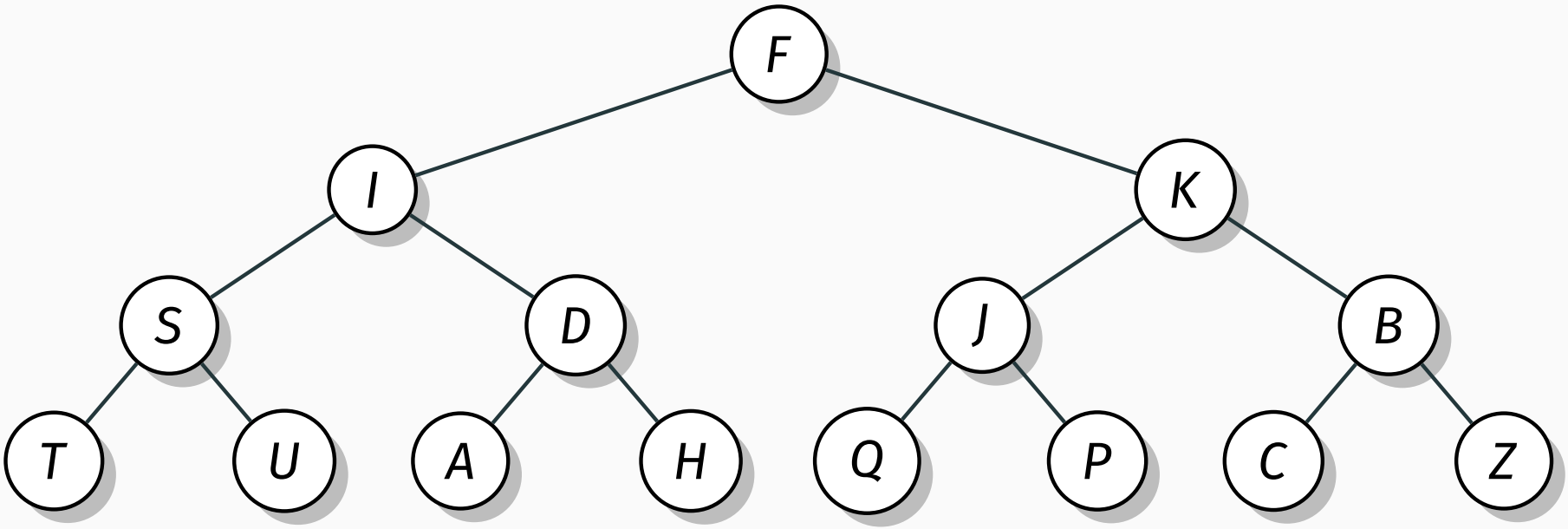
Binární strom – grafické znázornění rekurzivní definice



Binární strom, příklad



Úplný binární strom

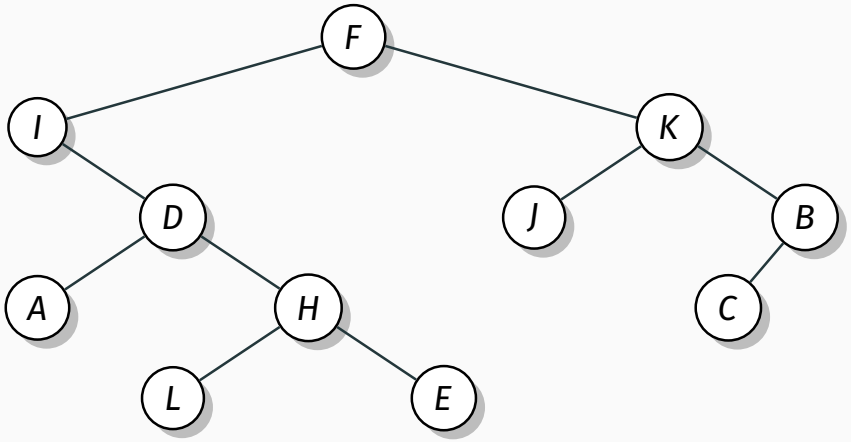


Úplný binární strom – každý vnitřní uzel má právě dva potomky.

Binární vyhledávací strom

Jak využít binární stromy jako datovou strukturu? Jakým způsobem v nich organizovat data?

Libovolně? Nesmysl – jde o zbytečně komplikovaný seznam!



Řešením je využít vlastností stromu (souvislost a jedinečnost cesty z uzlu do uzlu) a doplnit je vhodným „**navigačním pravidlem**“.

Binární vyhledávací strom – „navigační pravidlo“

Nechť y je uzel v binárním stromu. Potom pro každý uzel x v levém podstromu uzlu y a každý uzel z v pravém podstromu uzlu y platí

$$x_{key} < y_{key} < z_{key}.$$

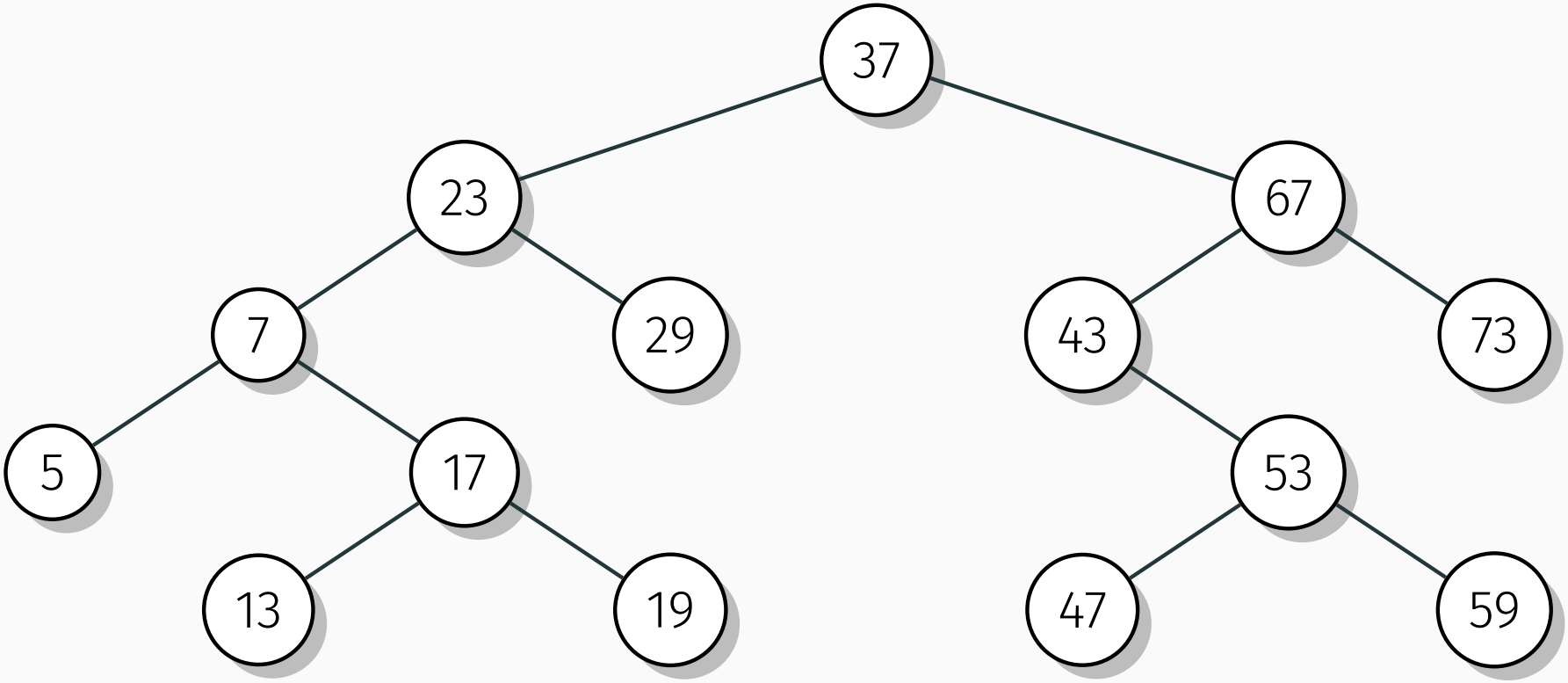
Binární strom, ve kterém pro všechny jeho uzly platí toto pravidlo nazýváme **binární vyhledávací strom** (angl. binary search tree).

Binární vyhledávací strom – „navigační pravidlo“ (pokrač.)

Poznámky

- Navigační pravidlo tedy určuje, jak mají být data v binárním vyhledávacím stromu rozmístěna.
- Znalosti rozmístění dat ve stromu využijeme při jejich vyhledávání.
- Algoritmy pro vložení a vyjmutí ze stromu jsou navázány na algoritmus vyhledávání.
- Binární vyhledávací strom je tedy už od počátku budován s ohledem na toto pravidlo.

Binární vyhledávací strom



Binární vyhledávací strom – vyhledávání

Hledání hodnoty a zahájíme v kořeni stromu r. Potom mohou nastat tyto možnosti:

1. Strom s kořenem r je prázdný, potom tento strom nemůže obsahovat uzel s klíčem a a hledání končí neúspěchem.
2. V opačném případě srovnáme klíč a s klíčem kořene r. V případě, že
   - 2.1 $a = r_{key}$ strom obsahuje uzel s klíčem a a hledání končí úspěšně;
   - 2.2 $a < r_{key}$ všechny uzly s klíči menšími než r_{key} jsou levém podstromu, pokračujeme rekurzivně v levém podstromu;
   - 2.3 $a > r_{key}$ všechny uzly s klíči většími než r_{key} jsou pravém podstromu, pokračujeme rekurzivně v pravém podstromu.

Binární vyhledávací strom

Efektivita mnoha algoritmů pracujících obecně s binárními stromy, např. vyhledávání v binárním vyhledávacím stromu závisí na výšce binárního stromu.

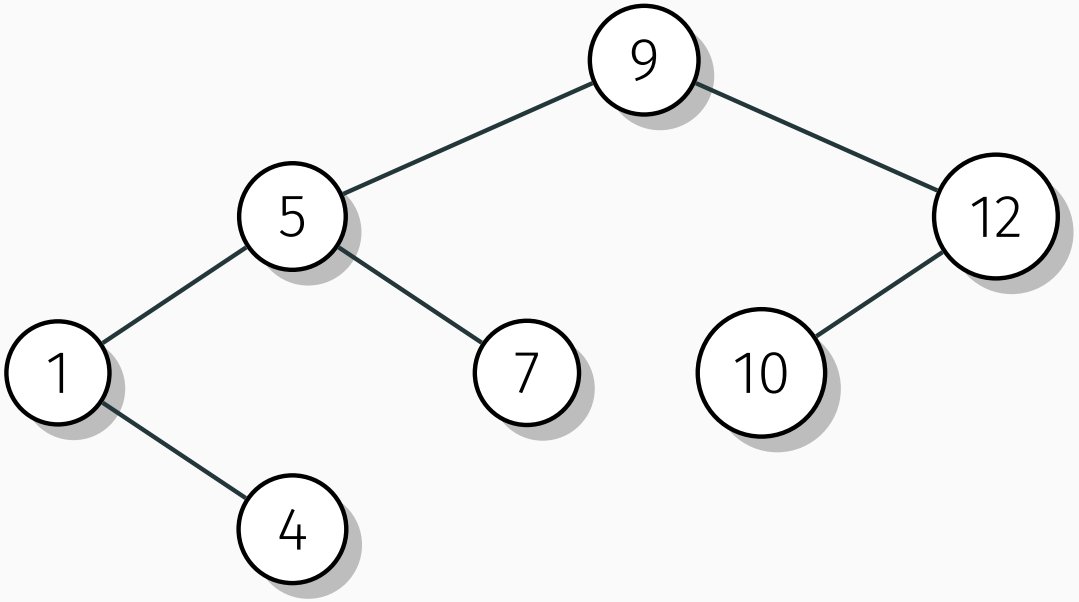
Pro výšku h binárního stromu s n uzly platí nerovnost

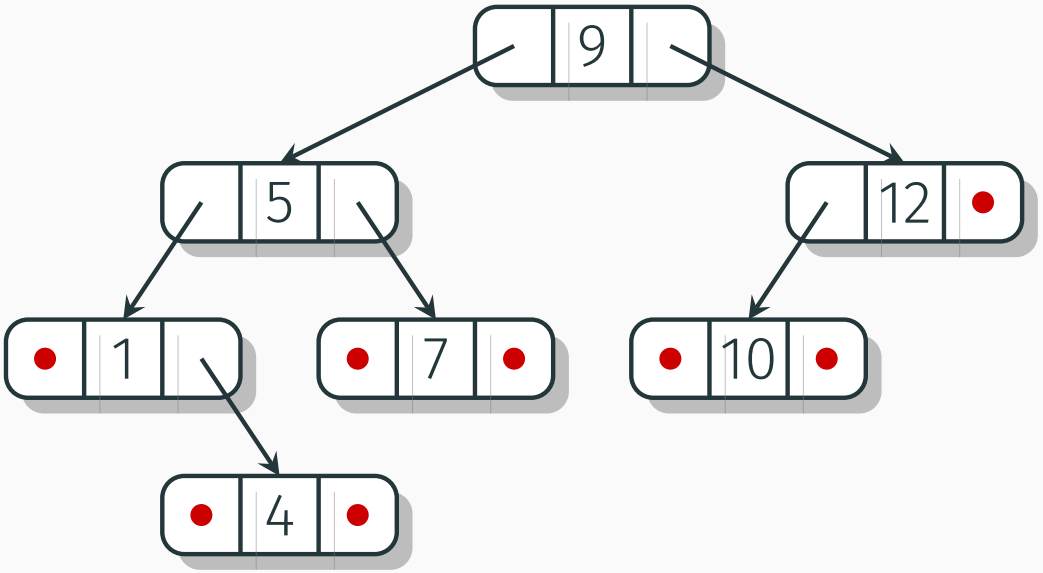
$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Binární vyhledávací strom – vkládání

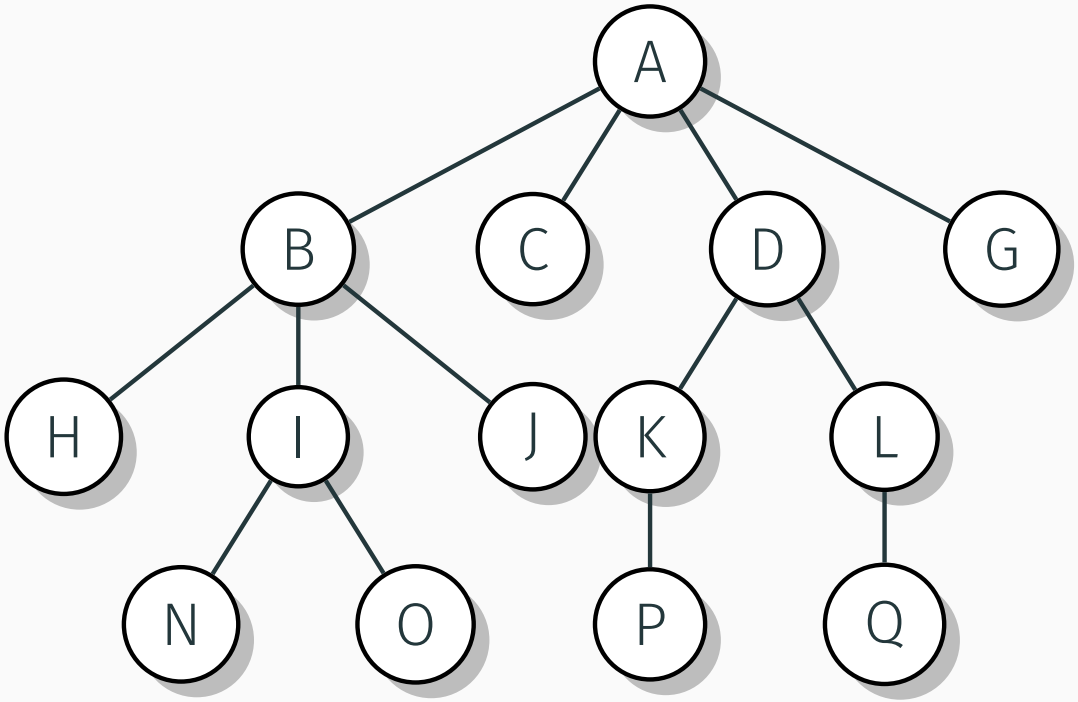
- Vložení klíče musí korespondovat s vyhledávacím algoritmem.
- Nejdříve se musíme pokusit vkládaný klíč ve stromu najít.
- Pokud jej nenajdeme, tak místo, kde jsme hledání neúspěšně zakončili odpovídá místu ve stromu, kde by tento klíč měl být.
- Toto plyne z jednoznačnosti cesty mezi kořenem a kterýmkoliv uzlem.
- Nový uzel je připojen jako nový list ke stromu – strom roste prostřednictvím listů.
- Otázkou je co s duplicitami? Řešení závisí na povaze konkrétní řešené úlohy.

Binární strom – standardní implementace



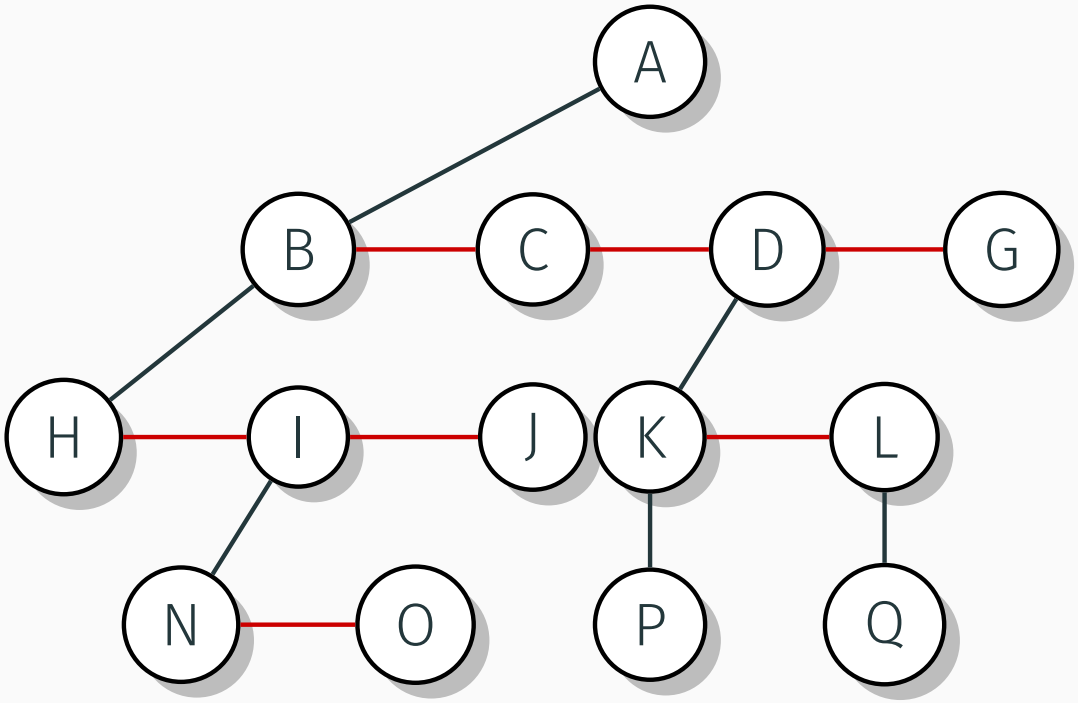


Reprezentace seřazeného stromu



- Každý uzel může mít libovolný počet potomků.
- Komplikovaná reprezentace uzlu – seznam potomků?

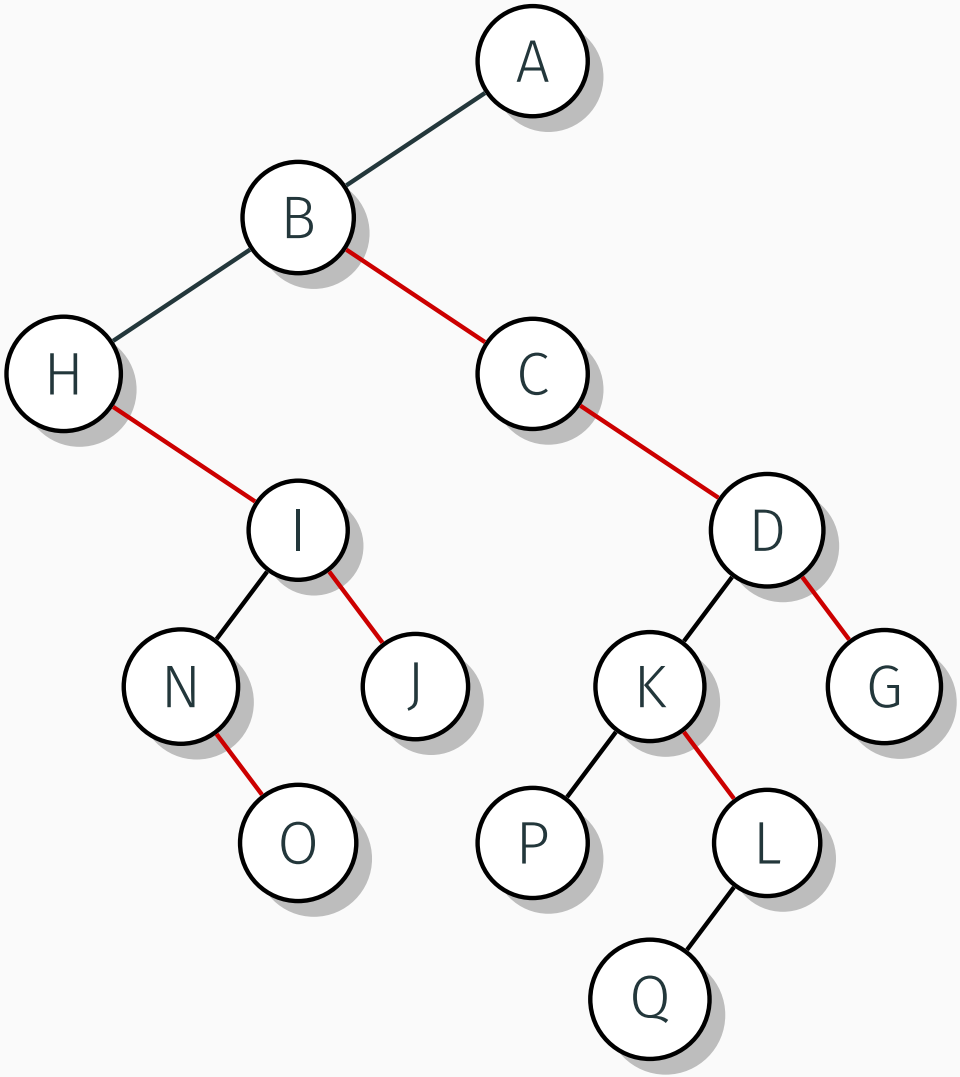
Reprezentace seřazeného stromu – first child – next sibling



First child – next sibling reprezentace – každý uzel obsahuje dva ukazatele:

1. ukazatel na prvního potomka a
2. ukazatel na sourozence.

Reprezentace seřazeného stromu – Knuthova transformace



First child – next sibling reprezentace otočena o **45**° po směru hodinových ručiček.

Množina

- Datovou strukturu množina budeme chápat jako nesetříděnou kolekci (i prázdnou) navzájem různých prvků.
- Množinu můžeme zadat dvěma způsoby:
  1. **výčtem prvků**, např. $M = \{2, 3, 5, 7\}$ nebo
  2. **vlastností**, které musí prvky splňovat, např. $M = \{n,$ prvočísla menší než $10\}$.
- Nejdůležitější množinové operace:
  - příslušnost prvku do množiny, čili dotaz „Je x prvkem M?“,
  - sjednocení dvou množin a
  - průnik dvou množin.

Množina – implementace

Bitový vektor

- univerzum $U = \{u_0, u_1, \dots, u_{n-1}\}$, $|U| = n$
- libovolnou množinu M považujeme za podmnožinu univerza U
- bitový vektor $\vec{b}$ dimenze n, kde

$$\vec{b}_i = \begin{cases} 1 & u_i \in M \\ 0 & \text{jinak} \end{cases}$$

Příklad

$U = \{0, 1, 2, \dots, 8, 9\}$

$M = \{2, 3, 5, 7\}$

$\vec{b} = (0, 0, 1, 1, 0, 1, 0, 1, 0, 0)$

Množina – implementace

Výčet prvků

- množinu reprezentujeme výčtem, prvků v ní obsažených
- podle okolností můžeme pro uložení prvků využít pole, spojový seznam, binární vyhledávací strom, hašovací tabulku atd.
- vždy záleží na konkrétním problému, jaké operace jsou podstatné, je-li podstatné udržovat uspořádané a tak dále

Slovník

- Pokud je s prvkem množiny svázán nějaký další údaj, mluvíme pak o **slovníku** (angl. dictionary, associative array, map, symbol table).
- Slovník udržuje dvojice (klíč, hodnota), kde klíč musí být ve slovníku unikátní.
- Matematicky jde o **zobrazení**.
- Nejdůležitější operace:
  - vložení dvojice do slovníku
  - smazání dvojice ze slovníku
  - modifikace hodnoty ve slovníku
  - vyhledání hodnoty pro daný klíč
- Pro implementaci lze využít pole, spojový seznam, binární vyhledávací strom, hašovací tabulku atd.

Děkuji za pozornost