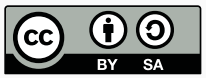
VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Algoritmy – prezentace k přednáškám

doc. Mgr. Jiří Dvorský, Ph.D.

Stav prezentace ke dni 28. dubna 2024

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Aktuální verze prezentací

Prezentace jsou průběžně, podle potřeb výuky, doplňovány a aktualizovány. Aktuální verzi prezentací najdete vždy na webu předmětu

www.cs.vsb.cz/dvorsky/Algorithms_Slides.html

Algoritmy I

Úvodní přednáška předmětu

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

Analýza složitosti algoritmů

Strategie řešení problémů hrubou silou a úplným prohledáváním

Stručná osnova všech přednášek (pokrač.)

Strategie řešení sniž a vyřeš

Snížení o konstantní faktor

Snížení o proměnný faktor

Strategie řešení rozděl a panuj

Algoritmy II

Úvodní přednáška předmětu

Stručná osnova všech přednášek (pokrač.)

Strategie řešení transformuj a vyřeš

Záměna paměťové a časové složitosti

Dynamické programování

Hladové algoritmy

Strategie řešení iterativním zlepšováním

Stručná osnova všech přednášek (pokrač.)

Meze možností algoritmického řešení problémů. P, NP a NP-úplné problémy.

Zdolávání mezí možností algoritmického řešení problémů

Ostatní

Přílohy

Algoritmy I

Úvodní přednáška předmětu

- O předmětu Algoritmy I
- Prezenční forma studia
  - Výuka
  - Úkoly a jejich hodnocení
- Kombinovaná forma studia
  - Výuka
  - Úkoly a jejich hodnocení
- Software pro výuku

Celková osnova všech přednášek (pokrač.)

Studijní literatura

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

Celková osnova všech přednášek (pokrač.)

Celková osnova všech přednášek (pokrač.)

Celková osnova všech přednášek (pokrač.)

Celková osnova všech přednášek (pokrač.)

Snížení o proměnný faktor

Strategie řešení rozděl a panuj

- Násobení velkých celých čísel
- Strassenovo násobení matic
- Problém nejbližší dvojice bodů
- Konvexní obal množiny

Algoritmy II

Úvodní přednáška předmětu

- O předmětu Algoritmy II

Celková osnova všech přednášek (pokrač.)

Celková osnova všech přednášek (pokrač.)

- Jedinečnost prvků v poli
- Výpočet modu
- Vyhledávání

Gaussova eliminační metoda

- LU-rozklad matice
- Inverzní matice
- Determinant matice

Vyvážené vyhledávací stromy

- AVL stromy
- 2-3 stromy

Celková osnova všech přednášek (pokrač.)

Celková osnova všech přednášek (pokrač.)

- Minimální kostra grafu
  - Primův algoritmus
  - Kruskalův algoritmus
- Dijkstrův algoritmus
- Huffmanův kód

Strategie řešení iterativním zlepšováním

Meze možností algoritmického řešení problémů. P, NP a NP-úplné problémy.

Zdolávání mezí možností algoritmického řešení problémů

Ostatní

Celková osnova všech přednášek (pokrač.)

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Úvodní přednáška předmětu

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Úvodní přednáška předmětu

O předmětu Algoritmy I

Upozornění

Všechny aktuální informace k předmětu naleznete na

http://www.cs.vsb.cz/dvorsky/

Tato prezentace slouží jen pro účely úvodní přednášky a nebude dále aktualizována.

O předmětu Algoritmy I

- Náplní předmětu jsou základní strategie algoritmického řešení úloh (hrubá síla, rozděl a panuj atd.) a typické příklady jejich užití.
- Přednášky jsou zaměřeny na **teorii**.
- Cvičení jsou zaměřena na **implementaci** řešení problémů danou strategií v jazyce C resp. C++.
- Vazby na další předměty:
  - Úvod do programování – jazyk C,
  - Funkcionální programování – rekurze a
  - Objektově orientované programování – asi není třeba komentáře.

Rozsah předmětu, způsob zakončení

Rozsah předmětu

- výuka probíhá v letním semestru prvního ročníku bakalářského studia
- hodinová dotace:
  - 2 hodiny přednášky a 2 hodiny cvičení týdně v prezenční formě a
  - 6 tutoriálů v kombinované formě studia

Zakončení – klasifikovaný zápočet

- klasifikovaný zápočet není zkouška, řídí jinými pravidly,
- prostudujte si proto **Studijní a zkušební řád pro studium v bakalářských a magisterských studijních programech**, článek 12.

Garant předmětu

doc. Mgr. Jiří Dvorský, Ph.D.

Kancelář: EA441

Email: *jiri.dvorsky@vsb.cz*

Web: *www.cs.vsb.cz/dvorsky*

K čemu je garant předmětu?

Garant předmětu zodpovídá za průběh výuky celého předmětu, průběh cvičení, plnění úkolů na cvičeních a za korektní hodnocení úkolů. Problémy spojené se cvičeními řešte primárně se svým cvičícím. Nepodaří-li dosáhnout řešení problému s cvičícím obracejte se na garanta předmětu.

Prerekvizity

- Prerekvizity jsou souhrnem požadavků, které je nutné splnit, aby si student mohl zapsat předmět. Prerekvizity jsou buď formální nebo věcné.
- Formální prerekvizity – žádné
- Věcné prerekvizity:
  - znalosti z předmětu Úvod do programování,
  - středoškolské matematiky a
  - obecná orientace ve výpočetní technice.
- Předmět Algoritmy I je ale povinnou prerekvizitou navazujícího předmětu Algoritmy II.

Docházka

Přednášky

- účast na přednáškách je **výrazně doporučená**.

Cvičení

- jsou **povinná**,
- účast a aktivita na cvičeních jsou hodnoceny,
- je nutno získat dostatečné bodové hodnocení.

Studenti se specifickými nároky

Centrum Slunečnice FEI

- *http://slunecnice-fei.vsb.cz/*,
- poskytuje podporu zpřístupňující studium i pro studenty se specifickými nároky,
- lze získat, mimo jiné, zvýšenou časovou dotaci na úkoly.

Výzva

Je vysoce žádoucí, aby studenti, kteří dostanou tuto zvýšenou časovou dotaci, neprodleně kontaktovali svého cvičícího a garanta předmětu, abychom předešli případným problémům!

Individuální studijní plán

Individuální studijní plán umožňuje, v odůvodněných případech, individuální termíny pro plnění studijních povinností.

Výzva

- Je vysoce žádoucí, aby studenti, kteří získají individuální studijní plán, neprodleně kontaktovali svého cvičícího a garanta předmětu a dohodli se na individuálních termínech plnění úkolů.
- Individuální studijní plán neznamená naprostou libovůli v termínech.
- Individuální studijní plán už vůbec neznamená možnost vybrat si úkoly, které budu plnit a které ne.

Konzultace

- Pokud nebudete ve výuce něčemu rozumět, potřebujete s čímkoliv poradit nebo vyřešit nějaký problém s přednáškou, cvičeními, testy, Vaší absencí na výuce atd. je možné využít si domluvit **individuální konzultaci**.
- Konzultaci je nutné si domluvit předem, například mailem.
- Pokud potřebujete poradit s učivem, připravte si materiály, které jste si k tématu prostudovali, vypište si co je Vám jasné a kde jste se „zasekli“ a potřebujete poradit.
- Konzultací s vyučujícím nic neriskujete – maximálně se dozvíte co potřebujete.
- Přijďte se zeptat rovnou ke zdroji informací – internetová fóra jsou zaplevelena různými polopravdami i naprostými nesmysly.

Algoritmy I – výuka, úkoly a jejich hodnocení pro různé formy studia

- Prezenční a kombinované studium má specifickou formu výuky.
- Obě formy studia mají specifické podmínky pro splnění předmětu.
- Podle formy Vašeho studia se Vás týká pouze jedna ze dvou následujících částí prezentace.

Úvodní přednáška předmětu

Prezenční forma studia

Témata přednášek

1. Organizační informace k předmětu Algoritmy I
2. Úvod
   - 2.1 Co je to algoritmus
   - 2.2 Základy algoritmického řešení problémů
   - 2.3 Důležité typy problémů
   - 2.4 Fundamentální datové struktury
3. Analýza složitosti algoritmů
   - 3.1 Základy analýzy složitosti algoritmů
   - 3.2 Asymptotické notace složitosti
   - 3.3 Analýza nerekurzivních algoritmů
   - 3.4 Analýza rekurzivních algoritmů
4. Strategie řešení problémů hrubou silou a úplným prohledáváním

Témata přednášek (pokrač.)

Témata přednášek (pokrač.)

5.5 Hledání mediánu
5.6 Interpolační vyhledávání
5.7 Vyhledávání a vkládání do binárního vyhledávacího stromu

6. Strategie řešení rozděl a panuj
   6.1 Třídění sléváním
   6.2 QuickSort
   6.3 Průchody binárním stromem
   6.4 Násobení velkých celých čísel a násobení matic
   6.5 Problém nejbližší dvojice bodů
   6.6 Konvexní obal množiny

Výše uvedená **témata přednášek** budou pochopitelně rozdělena do jednotlivých přednášek na celý semestr.

Cvičení

- Přímá výuka ve cvičeních odpovídá přednáškám.
- Ve cvičeních pracují studenti pod vedením cvičícího na konkrétní implementaci příkladů v jazyce C++.
- Dále je také možné konzultovat s cvičícím probírané učivo.
- Rozdělení do cvičení tak, jak je uvedeno v informačním systému Edison, je nutné respektovat.
- Není možné překračovat kapacitu cvičení.
- Veškeré přesuny je nutné mít zaznamenány v systému Edison.

Cvičení (pokrač.)

- Cvičení nenahrazuje přednášku!
  - Účelem cvičení není příprava na závěrečnou písemku.
  - Cvičení nejsou bleskovou přednáškou pro ty, kteří nechodí na přednášky.
  - Na cvičení je nutné být připraven.

Úkoly

- Hodnocení v předmětu Algoritmy I se skládá ze tří částí, úkolů:
  1. **průběžné aktivity na cvičeních**,
  2. **obhajoby projektu** a
  3. **závěrečné písemné práce**.
- Všechny úkoly jsou povinné.
- Z každého úkolu je nutné získat aspoň minimální počet bodů.

Úkoly – průběžná aktivita na cvičeních

- Tato část hodnocení probíhá **průběžně po celý semestr**.
- Na každém cvičení bude cvičícím ohodnocena Vaše aktivita. Aktivita je hodnocena pomocí barevného kódu:
  - zelená – student na cvičení pracoval aktivně, v látce se orientoval, dařilo se mu implementovat zadané úkoly,
  - oranžová – student na cvičení byl spíše pasivní, na cvičení nebyl příliš připraven (ve znalostech měl „mezery“), implementace úkolů se příliš nedařila a
  - červená – student na cvičení byl zcela pasivní, o výuku nejevil zájem, implementaci úkolů nezvládl. Do této kategorie spadá i neomluvená neúčast na cvičení.

Úkoly – průběžná aktivita na cvičeních (pokrač.)

- Každému barevnému kódu odpovídá určitá váha, která se projeví v celkovém hodnocení všech cvičení. Zelená aktivita má váhu 1, oranžová má váhu 0,5 a červená 0.
- Z takto získaných vah z jednotlivých cvičení se na konci semestru vypočte průměrná váha, která se vynásobí maximálním možným počtem bodů (30) a výsledek je Vámi získaný počet bodů.
- Je zřejmé, že všechny zelené kódy odpovídají maximálnímu počtu bodů (30), samé červené kódy odpovídají nulovému počtu bodů.
- Body za aktivitu nelze získat zpětně.

Úkoly – průběžná aktivita na cvičeních (pokrač.)

Příklad

Student Franta na pěti cvičeních získal zelené hodnocení, na třech oranžové a na dvou červené. Průměrnou váhu vypočtene jako:

$$\frac{5 \times 1 + 3 \times 0,5 + 2 \times 0}{5 + 3 + 2} = \frac{6,5}{10} = 0,65.$$

Výsledné bodové hodnocení je tedy $0,65 \times 30 = 19,5$ bodů.

Úkoly – obhajoba projektu

- Zadání projektu bude zveřejněno na webu předmětu začátkem dubna 2024.
- Deadline pro odevzdání je 12. května 2024 23:59.
- Způsob odevzdání stanoví jednotliví cvičící.
- Obhajoby projektů proběhnou v zápočtovém týdnu a ve zkouškovém období. Harmonogram obhajob je v kompetenci cvičících.
- Bez ohledu na to, kdy proběhnou obhajoby projektů platí, že se obhajuje verze, která byla odevzdána do deadlinu.
- Na obhajobu projektu není možný opravný termín.

Úkoly – závěrečná písemná práce

- Závěrečná písemná práce se bude psát ve zkouškovém období.
- Všechny termíny budou vypsány v systému Edison.
- Opravný termín na závěrečnou písemnou práci je poskytován jen těm studentům, kteří na první pokus získali aspoň 10 bodů.

Počet bodů na prvním termínu	Opravný termín
0 až 9	NE
10 až 20	ANO
více než 21	není nutný, úspěch

Úkoly – závěrečná písemná práce (pokrač.)

- Závěrečnou písemnou práci máte možnost psát celkem **dvakrát**, jinak řečeno máte nárok na jednu opravu. Předmět je ukončen klasifikovaným zápočtem. Nevztahuje se tudíž na něj požadavek dvou oprav, jak to vyžaduje studijní řád u zkoušky.
- Žádné další opravy nejsou možné.

Hodnocení úkolů

- Je nutné splnit **všechny výše uvedené úkoly**,
- a zároveň u všech úkolů **aspoň minimální počet bodů**.

Úkol	Počet bodů	
	minimum	maximum
Průb. aktivita na cvičeních	15	30
Obhajoba projektu	15	30
Písemná práce	21	40
Celkový počet bodů	51	100

Úvodní přednáška předmětu

Kombinovaná forma studia

Tutoriály

1. tutoriál – 23. února 2024 povinný

- Na tomto úvodním tutoriálu Vám budou sděleny informace o organizaci studia předmětu a informace o náplni předmětu.
- Konzultace k tématům: Algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

2. tutoriál – 8. března 2024

- Konzultace k tématům: Analýza složitosti algoritmů.

Tutoriály (pokrač.)

3. tutoriál – 22. března 2024

- Konzultace k tématům: Strategie řešení problémů hrubou silou. Třídění výběrem, bublinové třídění. Sekvenční vyhledávání. Konvexní obal množiny bodů. Nalezení nejbližší dvojice bodů.

4. tutoriál – 12. dubna 2024

- Konzultace k tématům: Strategie řešení úplným prohledáváním. Problém obchodního cestujícího. Problém batohu. Průchody grafem.

Tutoriály (pokrač.)

5. tutoriál – 26. dubna 2024

- Konzultace k tématům: Strategie řešení sniž a vyřeš. Třídění vkládáním. Generování permutací a podmnožin. Vyhledávání půlením intervalu. Nalezení mediánu. Interpolační vyhledávání. Vyhledávání a vkládání do binárního vyhledávacího stromu.

6. tutoriál – 11. května 2024

- Konzultace k tématům: Strategie řešení rozděl a panuj. QuickSort. MergeSort. Konvexní obal množiny bodů. Nalezení nejbližší dvojice bodů.

Úkoly

- Hodnocení v předmětu Algoritmy I se skládá ze tří částí, úkolů:
  1. **průběžné aktivity na tutoriálech**,
  2. **obhajoby projektu** a
  3. **závěrečné písemné práce**.
- Všechny úkoly jsou povinné.
- Z každého úkolu je nutné získat aspoň minimální počet bodů.
- Další informace o jednotlivých úkolech budou k dispozici na webu tutora.

Úkoly – průběžná aktivita na tutoriálech

Průběžná aktivita na tutoriálech znamená:

- účast na tutoriálech a
- průběžné plnění úkolů zadaných na jednotlivých tutoriálech.

Úkoly – obhajoba projektu

- Zadání projektu bude zveřejněno na webu předmětu začátkem dubna 2024.
- Deadline pro odevzdání je 12. května 2024 ve 23:59.
- Způsob odevzdání a další náležitosti budou upřesněny na webu tutora.
- Obhajoby projektů proběhnou ve zkouškovém období. Termíny budou vypsány v systému Edison.
- Bez ohledu na to, kdy proběhnou obhajoby projektů platí, že se obhajuje verze, která byla odevzdána do deadlinu.
- Na obhajobu projektu není možný opravný termín.

Úkoly – závěrečná písemná práce

- Závěrečná písemná práce je zaměřena na teoretické znalosti.
- Závěrečné písemná práce proběhne ve zkouškovém období.
- Termíny budou vypsány v systému Edison.
- Opravný termín na závěrečnou písemnou práci je poskytován jen těm studentům, kteří na první pokus získali aspoň 10 bodů.

Počet bodů na prvním termínu	Opravný termín
0 až 9	NE
10 až 20	ANO
více než 21	není nutný, úspěch

Úkoly – závěrečná písemná práce (pokrač.)

- Závěrečnou písemnou práci máte možnost psát celkem **dvakrát**, jinak řečeno máte nárok na jednu opravu. Předmět je ukončen klasifikovaným zápočtem. Nevztahuje se tudíž na něj požadavek dvou oprav, jak to vyžaduje studijní řád u zkoušky.
- Žádné další opravy nejsou možné.

Hodnocení úkolů

- Je nutné splnit **všechny výše uvedené úkoly**,
- a zároveň u všech úkolů **aspoň minimální počet bodů**.

Úkol	Počet bodů	
	minimum	maximum
Průb. aktivita na tutoriálech	15	30
Obhajoba projektu	15	30
Písemná práce	21	40
Celkový počet bodů	51	100

Úvodní přednáška předmětu

Software pro výuku

Software pro výuku

Primární software

- Vývojové prostředí pro C++
- Dokumentace k C++

Doplňkový software

- Dokumentační systém Doxygen, *www.doxygen.org*
- Typografický systém LaTeX, *www.ctan.org*

Vývojové prostředí pro C++

- Na učebnách je pro výuku k dispozici Microsoft Visual Studio Community 2022.
- Toto vývojové prostředí doporučuji i pro domácí přípravu.
- Obecně lze použít jakékoliv vývojové prostředí s kompilátorem podporujícím minimálně specifikaci **C++17**.

Vývojové prostředí pro C++ (pokrač.)

Poznámky

1. Při hodnocení Vašich projektů bude používán překladač **Microsoft Visual C++** a specifikace jazyka **C++17**.
2. Pozor na nestandardní rozšíření jazyka C++ implementovaného v GNU C++ kompilátoru.
3. Jazyk C není totožný s jazykem C++!

Úvodní přednáška předmětu

Studijní literatura

Studijní literatura

Studijní literaturu lze rozdělit do dvou skupin:

- **povinná literatura** – strategie algoritmického řešení problémů a
- **doporučená literatura** – programovací jazyk C++.

Níže uvedenou literaturu využijete v předmětu Algoritmy I i Algoritmy II.

Povinná literatura

1. LEVITIN, Anany. *Introduction to the Design and Analysis of Algorithms*. 3rd ed. Boston: Pearson, 2012. ISBN 978-0-13-231681-1.
2. CORMEN, Thomas H., Charles Eric LEISERSON, Ronald L. RIVEST a Clifford STEIN, [2022]. *Introduction to algorithms*. Fourth edition. Cambridge, Massachusetts: The MIT Press. ISBN 978-026-2046-305.
3. SEDGEWICK, Robert. *Algoritmy v C*. Praha: SoftPress, 2003. ISBN 80-864-9756-9.

Povinná literatura (pokrač.)

4. MAREŠ, Martin a Tomáš VALLA, 2017. *Průvodce labyrintem algoritmů* [online]. Praha: CZ.NIC, z.s.p.o. [cit. 2020-10-03]. CZ.NIC. ISBN 978-80-88168-19-5. Dostupné z: *https://knihy.nic.cz/*
5. WRÓBLEWSKI, Piotr. *Algoritmy*. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.
6. WIRTH, N. *Algoritmy a štruktúry údajov*. Alfa, Bratislava 1989.

Doporučená literatura

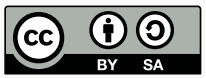
1. STROUSTRUP, Bjarne. *C++ programovací jazyk*. Praha: Softwarové Aplikace a Systémy, 1997. ISBN 80-901-5072-1.
2. VIRIUS, Miroslav. *Pasti a propasti jazyka C++*. 2., aktualiz. a rozš. vyd. Brno: CP Books, 2005. ISBN 80-251-0509-1.
3. SCHILDT, Herbert. *Nauč se sám C++: [poznej, vyzkoušej, používej]*. Praha: SoftPress, 2001. ISBN 80-864-9713-5.
4. ECKEL, Bruce. *Myslíme v jazyku C++*. Praha: Grada, 2000. Knihovna programátora (Grada). ISBN 80-247-9009-2.

Děkuji za pozornost

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

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



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

Co je to algoritmus?

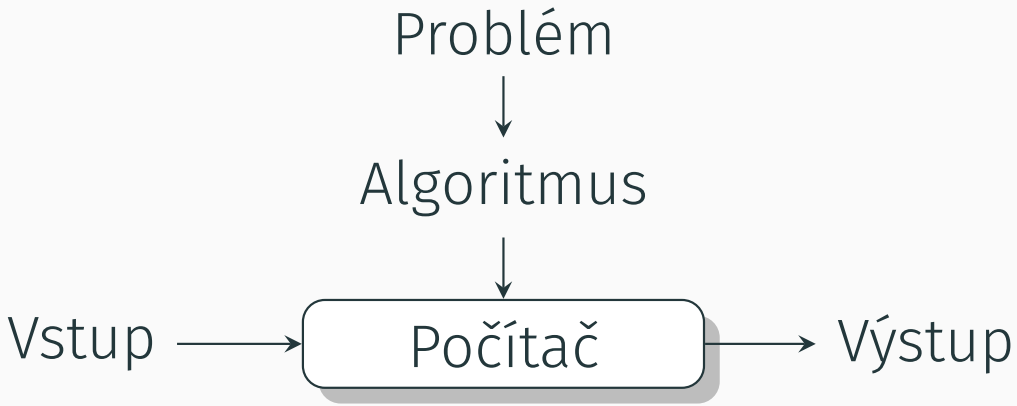
Proč studovat algoritmy?

- 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.

Největší společný dělitel – 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.

Největší společný dělitel – Euklidův algoritmus (pokrač.)

Příklad

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

Největší společný dělitel – 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.

Největší společný dělitel – 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, $\gcd(m, n)$

```
1 while n ≠ 0 do
2 |   r ← m mod n;
3 |   m ← n;
4 |   n ← r;
5 end
6 return m;
```

Největší společný dělitel – 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 $t = 12$.

Největší společný dělitel – 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.

Největší společný dělitel – 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!**

Největší společný dělitel – 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.

Největší společný dělitel – algoritmus rozkladem na prvočinitele (pokrač.)

Příklad

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

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

Největší společný dělitel – 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 $\leq n$

```
1  for p ← 2 to n do
2  |   A[p] ← p;
3  end
4  for p ← 2 to ⌊√n⌋ do
5  |   if A[p] ≠ 0 then          // p nebylo dosud vyloučeno
6  |   |   j ← p²;
7  |   |   while j ≤ n do
8  |   |   |   A[j] ← 0;
9  |   |   |   j ← j + p;
10 |   |   end
11 |   end
12 end
```

Eratosthenovo síto (pokrač.)

```
13 // Čísla, která nebyla z pole A vyloučena, okopírujeme
      do pole L
14 i ← 0;
15 for p ← 2 to n do
16 |   if A[p] ≠ 0 then
17 |   |   L[i] ← A[p];
18 |   |   i ← i + 1;
19 |   end
20 end
```

Největší společný dělitel – 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…

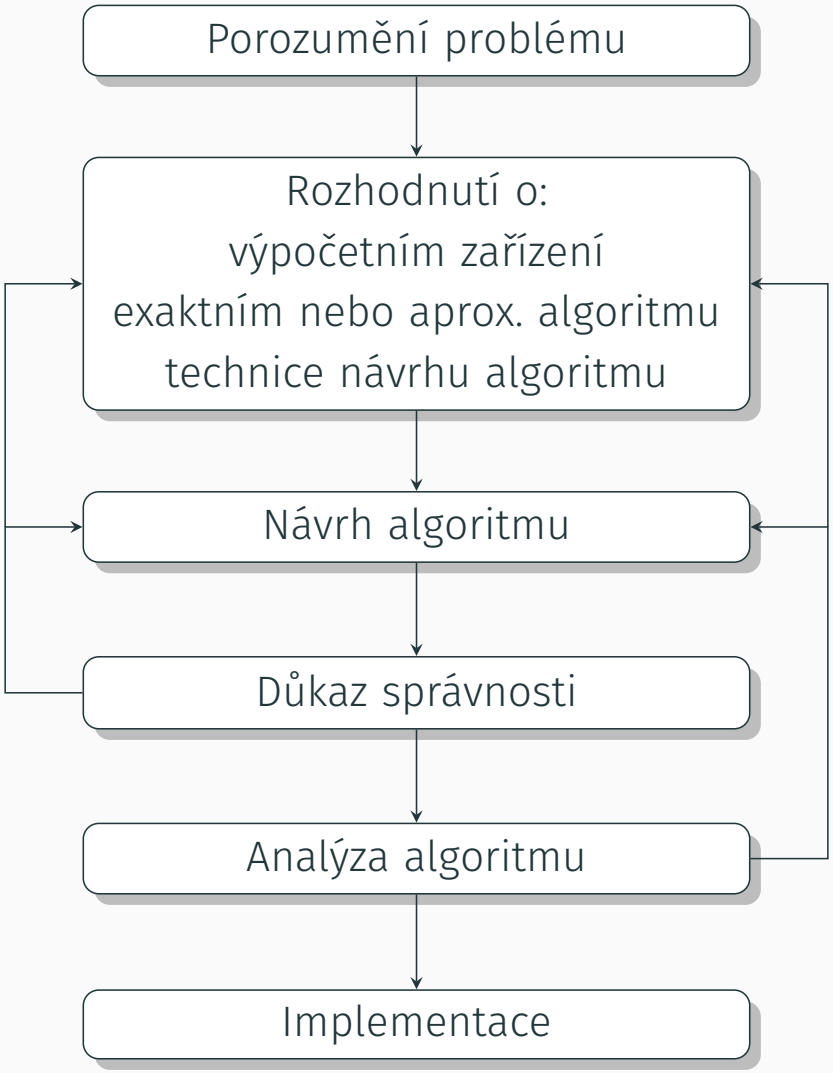
Co je to algoritmus. Strategie řešení problémů pomocí algoritmů. Významné typy řešených problémů.

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.

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.

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“

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.)

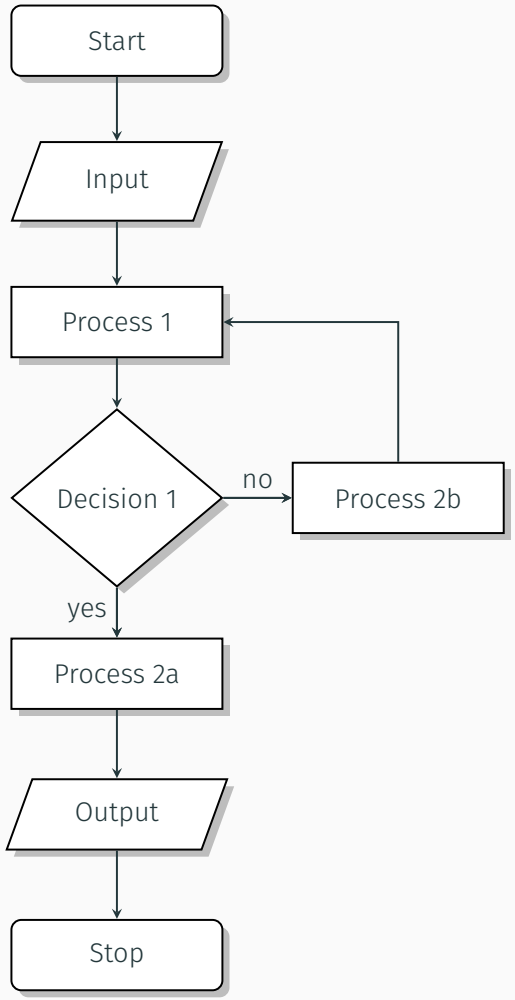
Programovací jazyk

- 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

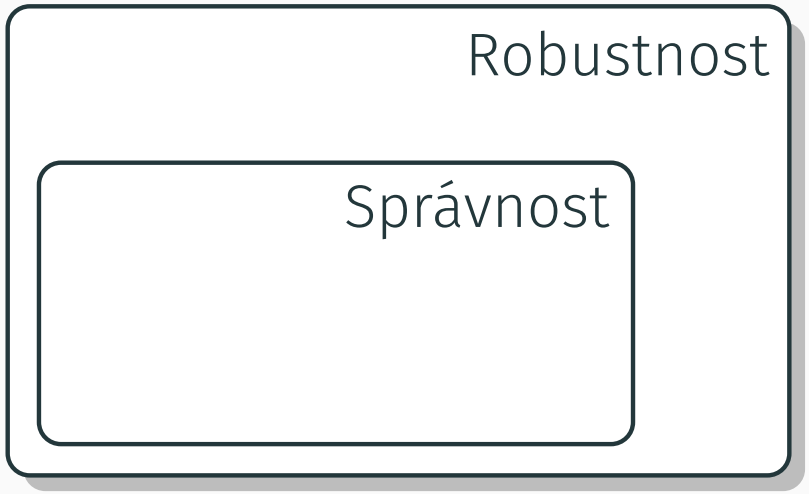
Kódování algoritmu

- 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 $\rho \subseteq A \times A$ $\forall x, y, z \in A$:

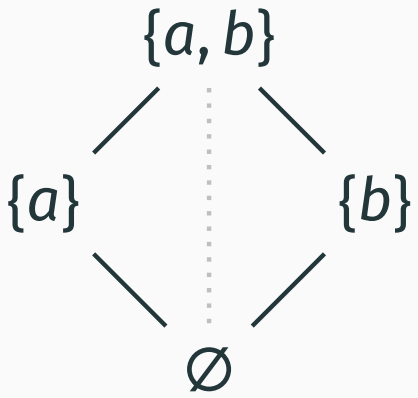
- reflexivita: $x \rho x$,
- ireflexivita: $\neg(x \rho x)$,
- asymetrie: $x \rho y \Rightarrow y \rho x$,
- antisymetrie: $x \rho y \wedge y \rho x \Rightarrow x = y$,
- tranzitivita: $x \rho y \wedge y \rho z \Rightarrow x \rho z$,
- souvislost: $[x \neq y \Rightarrow x \rho y \vee y \rho x]$,
- úplnost: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

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



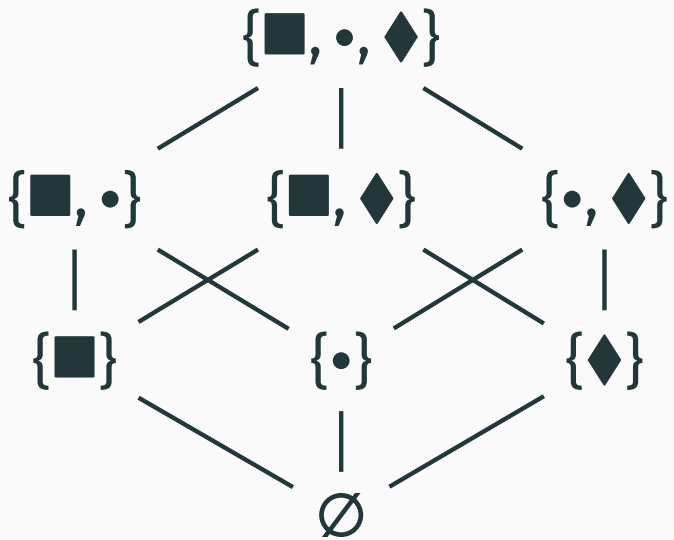
Hasseův diagram pro relaci uspořádání „být podmnožinou“ na množině $\{a, b\}$. Tranzitivní hrana, která se běžně nezobrazuje, je zobrazena tečkovaně.

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 $A = a_1, a_2, \ldots a_n$.
  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

$$a_{\pi_1} < a_{\pi_2} < a_{\pi_3} < a_{\pi_4} < a_{\pi_5} < a_{\pi_6}$$

$$a_6 < a_2 < a_4 < a_5 < a_1 < a_3$$

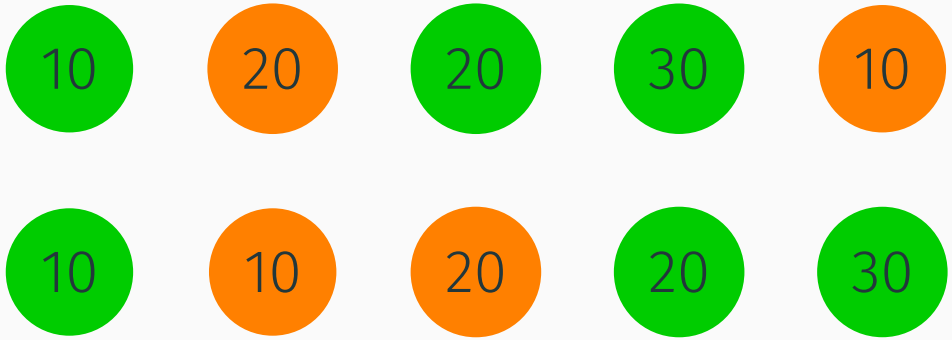
$$a < b < c < d < e < f$$

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 $1 \ldots n$.
- 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	$n-1, n-2, n-3, \ldots, 1$	$n-1$
$n-1$	$n-2, n-3, \ldots, 1$	$n-2$
$\vdots$	$\vdots$	$\vdots$
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

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

Odtud

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

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 vzorku	Předzpracování textu: ne	Předzpracování textu: ano
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“

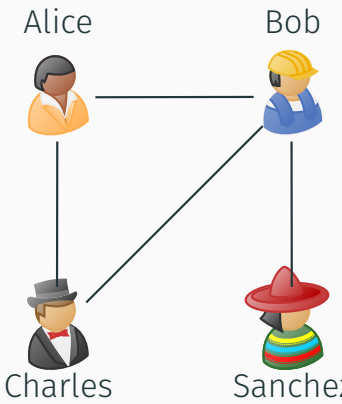
- sousměrné algoritmy – vzorek je procházen stejným směrem
- protisměrné algoritmy – vzorek je procházen opačným směrem.

Vyhledávání v textu – označení

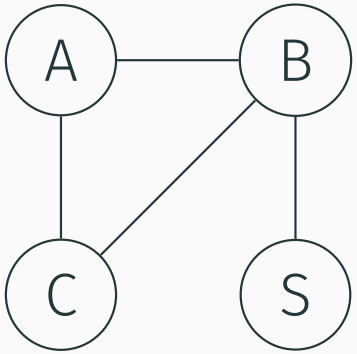
V dalším textu budeme používat následující označení:

- p hledaný vzorek, $p = p_0 p_1 \ldots p_{m-1}$, kde $|p| = m$ je délka vzorku,
- t prohledávaný text, $t = t_0 t_1 \ldots t_{n-1}$, 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)

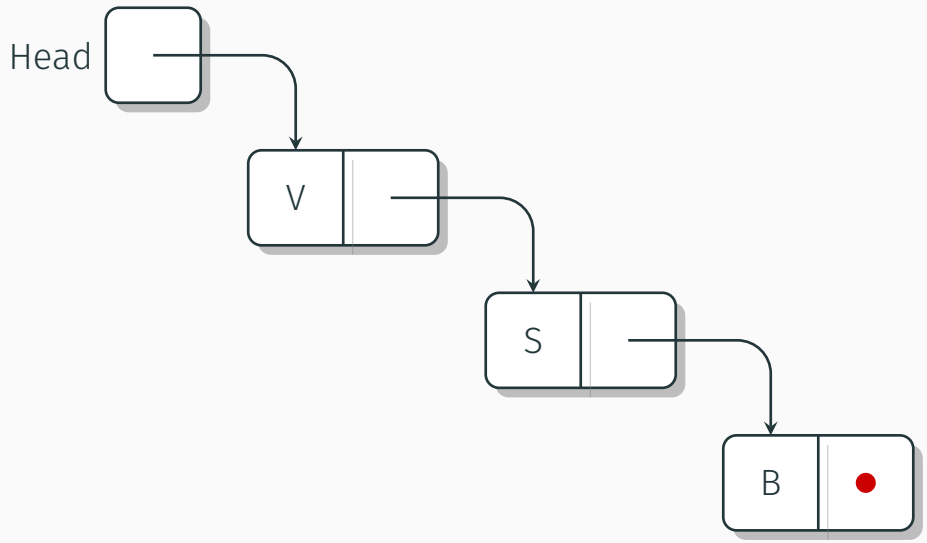
- Konečná posloupnost n hodnot uložených ve spojitém úseku paměti.
- Přístup pomocí indexu, náhodný přístup s konstantní časovou složitostí.
- Index:
  - nezáporné celé číslo,
  - pole s n hodnotami má rozsah indexů vždy $0, \ldots, n - 1$

- 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.)

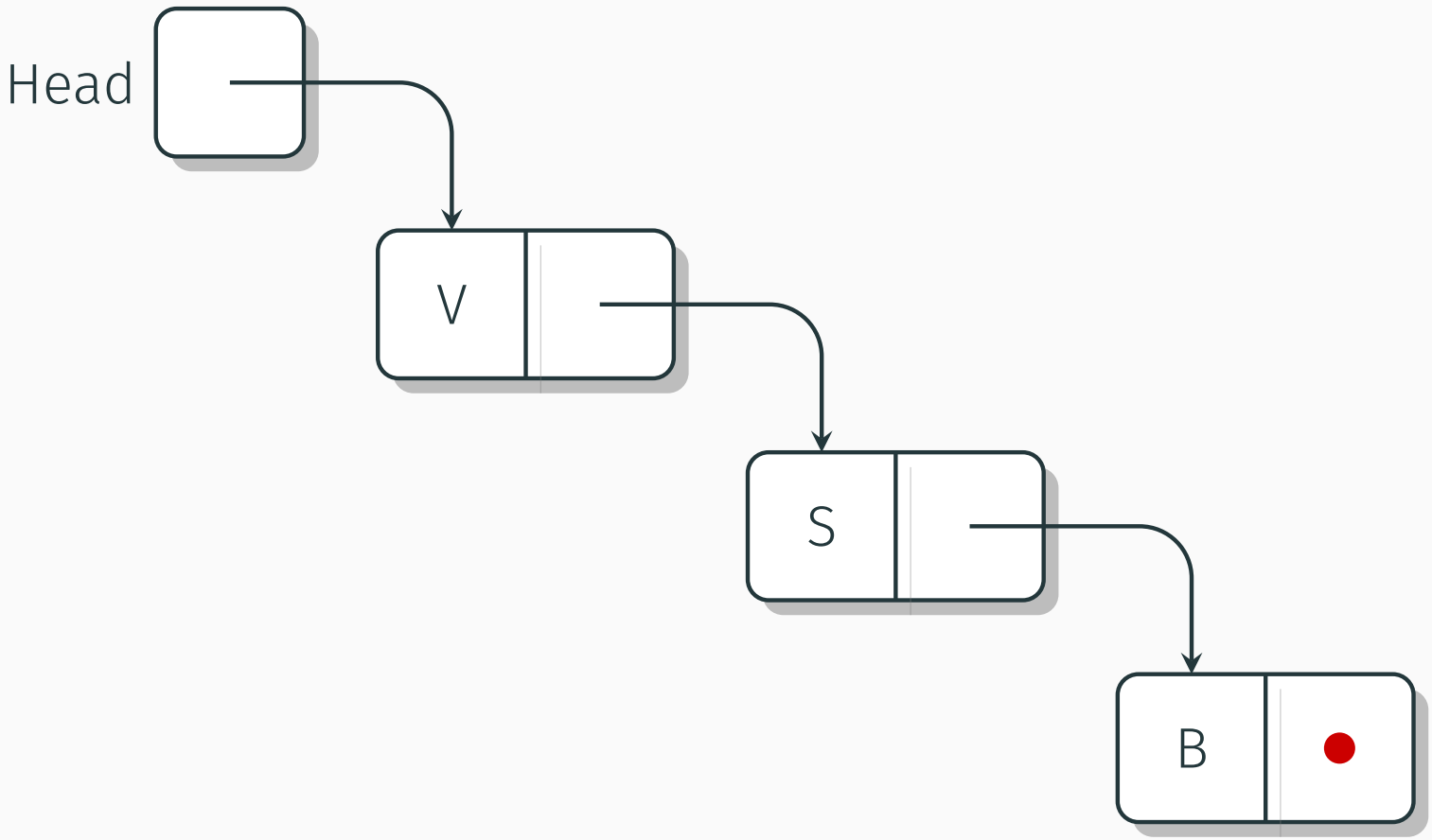
Varianty seznamu

- **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.

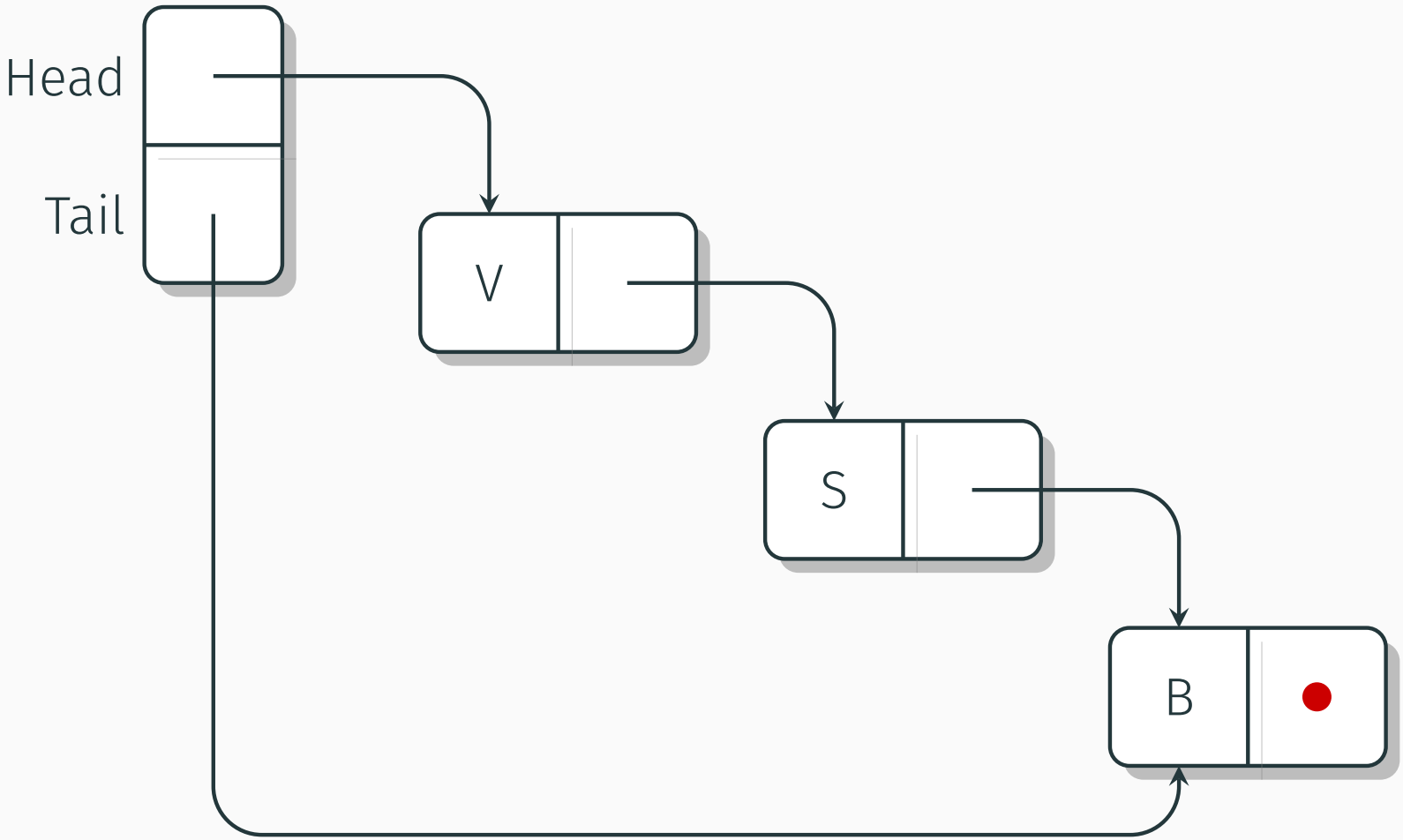
Seznam – jednosměrný seznam

- 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**.

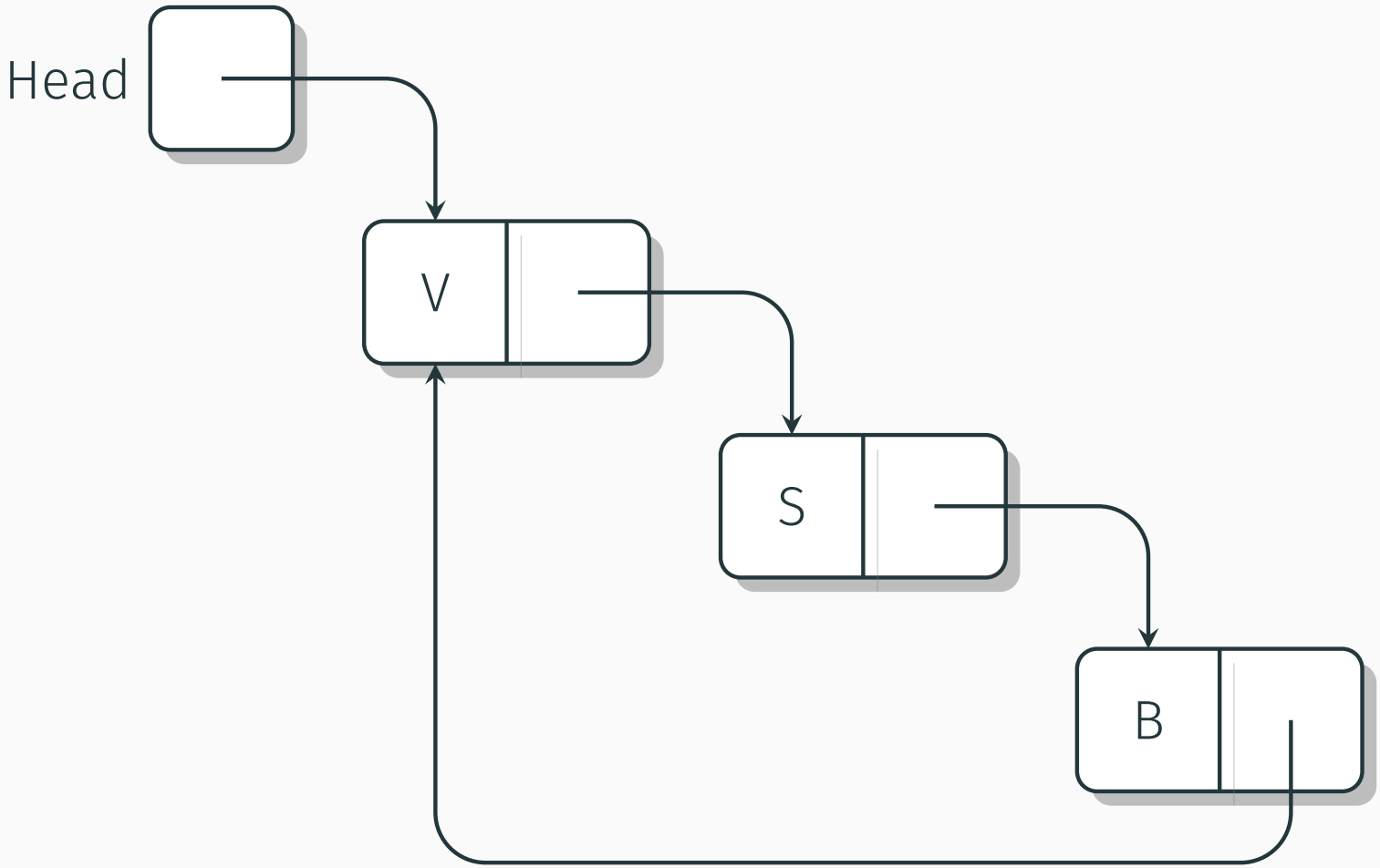
Seznam – jednosměrný, jen hlava seznamu



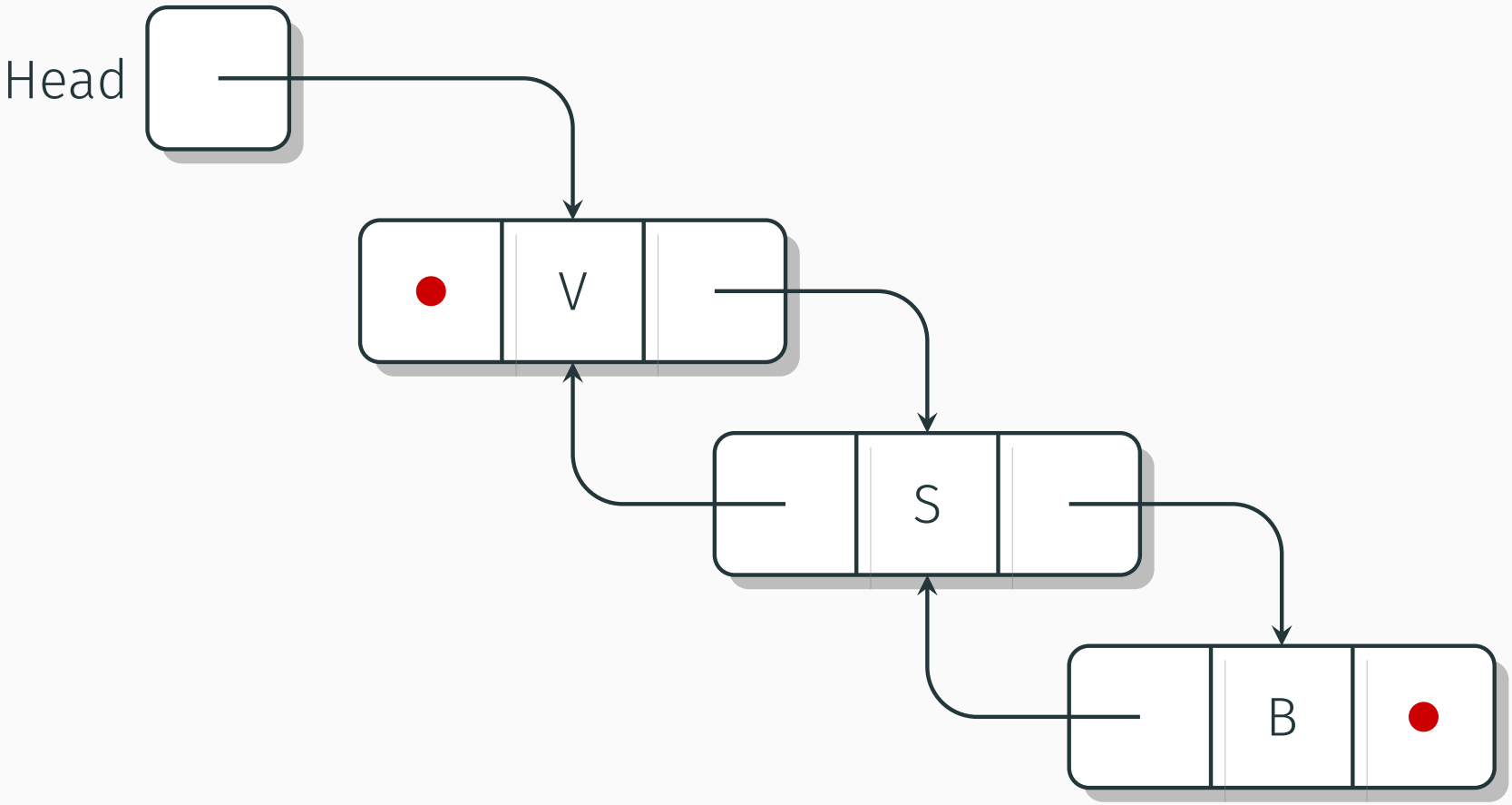
Seznam – jednosměrný, hlava i ocas seznamu



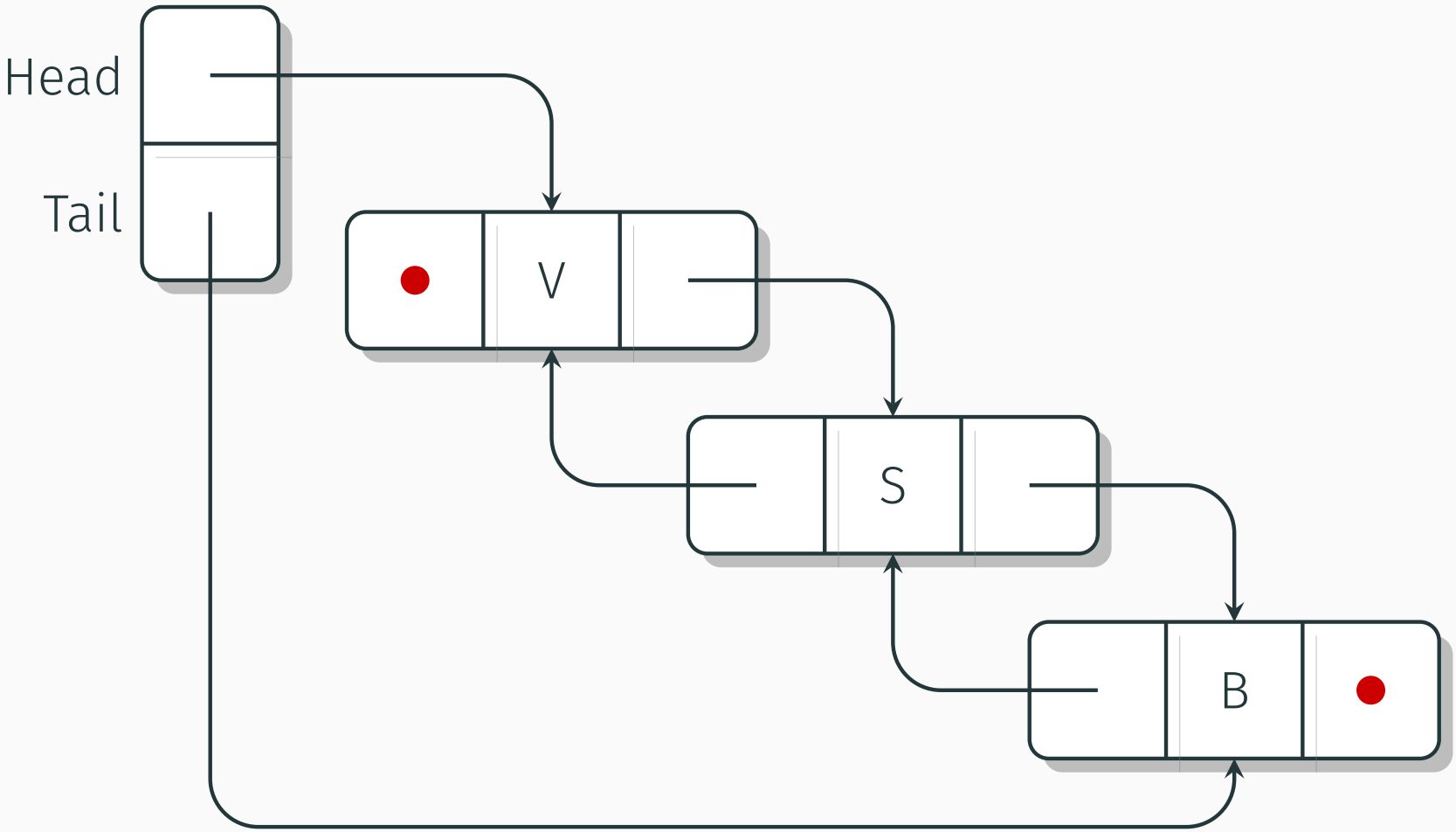
Seznam – jednosměrný, kruhový



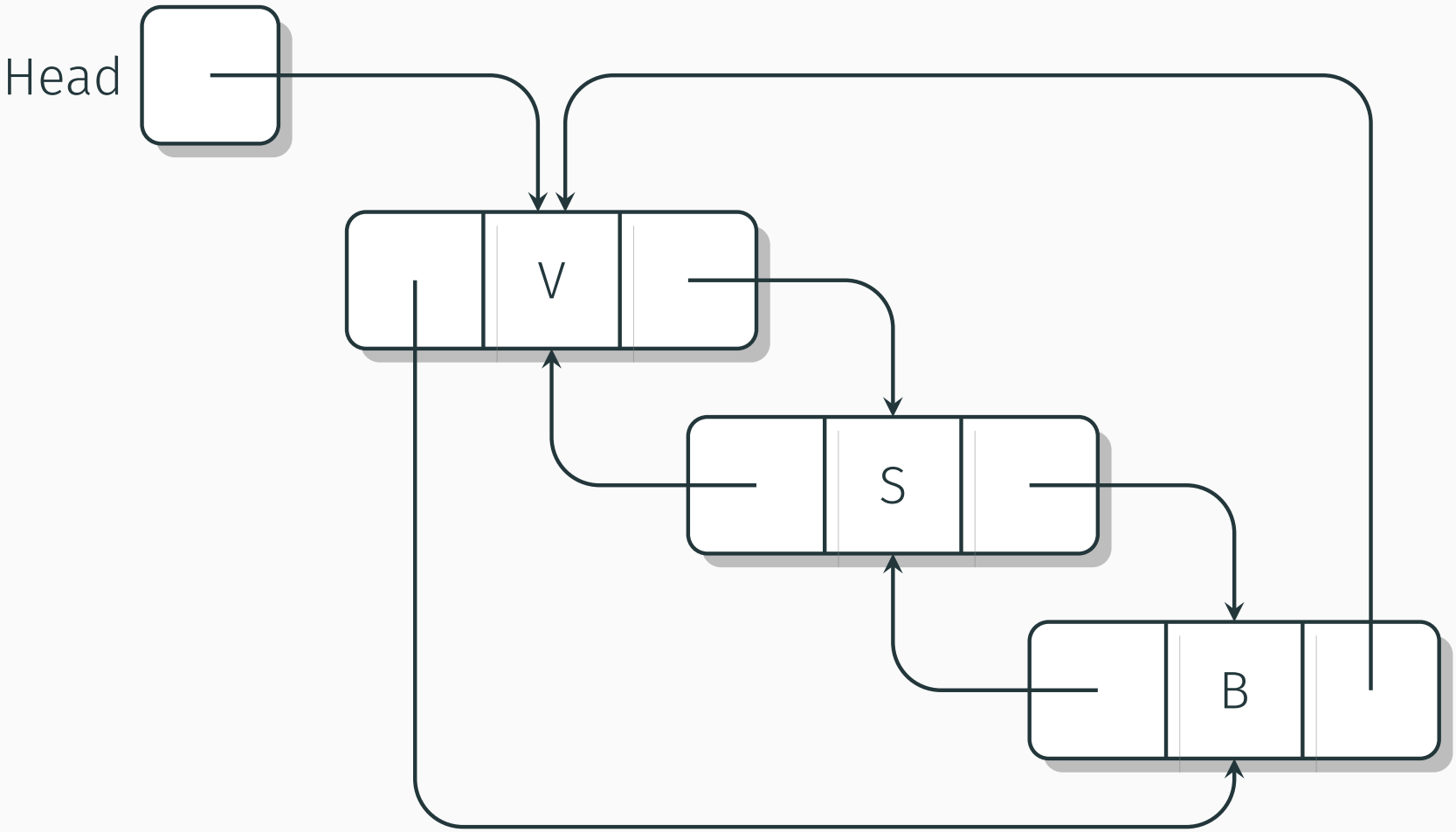
Seznam – obousměrný, jen hlava seznamu



Seznam – obousměrný, hlava i ocas seznamu



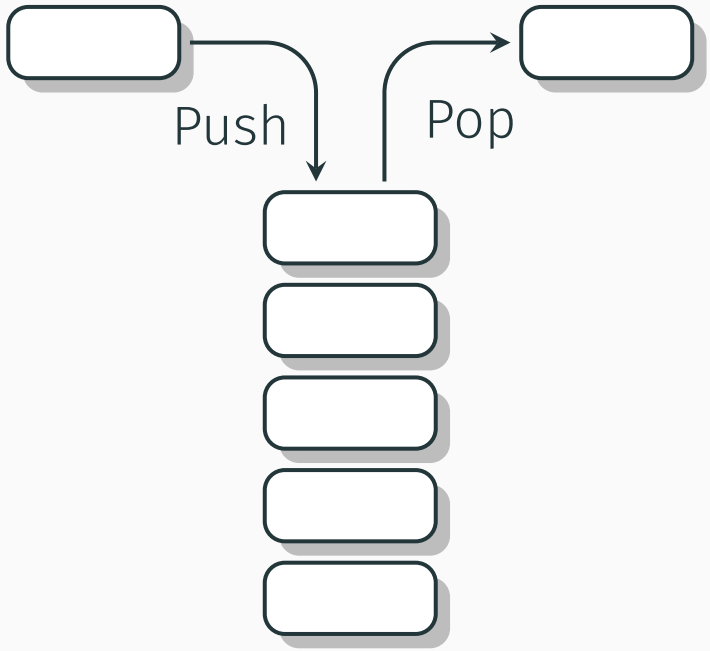
Seznam – obousměrný, kruhový



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

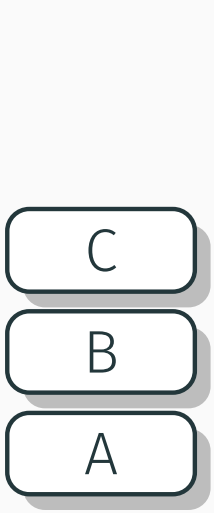
- Push – vložení prvku na vrchol zásobníku
- Pop – vyjmutí prvku z vrcholu zásobníku
- IsEmpty – test prázdnosti zásobníku
- Top – vrátí prvek z vrcholu zásobníku bez jeho vyjmutí

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).

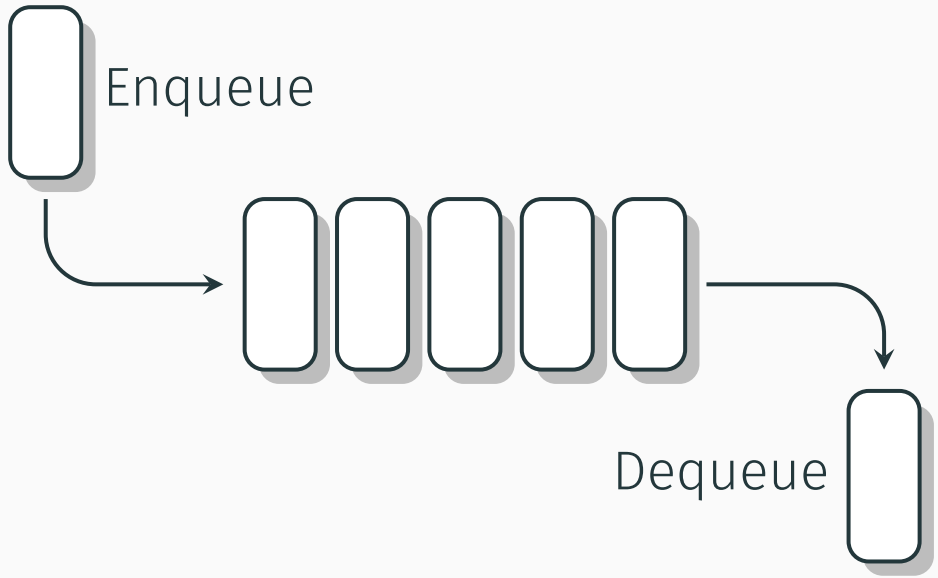
Zásobník – využití zásobníku

- 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

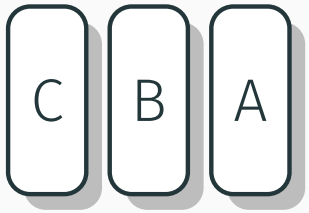
- Enqueue – vložení prvku na konec fronty
- Dequeue – vyjmutí prvku ze začátku fronty
- Peek – vrátí prvek ze začátku fronty bez jeho vyjmutí
- IsEmpty – test zda je fronta prázdná

Další možné operace

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

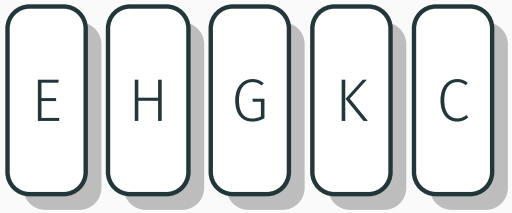
Správně implementované operace mají konstantní časovou složitost $O(1)$, tj. jejich časová složitost nezávisí na počtu prvků ve frontě.

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).

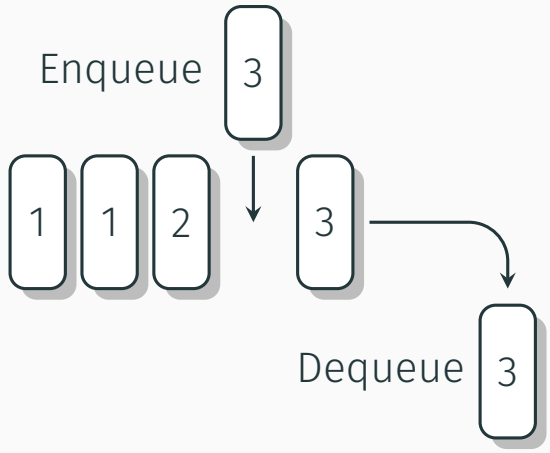
Fronta – využití

- 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



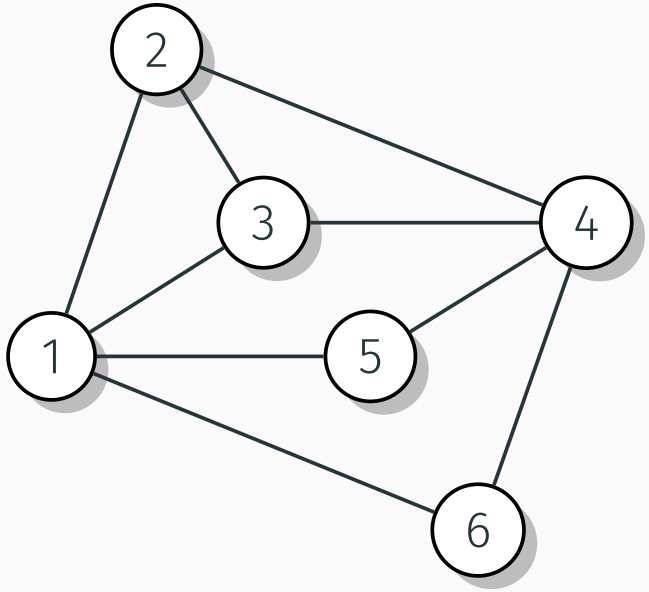
Implementace

- 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

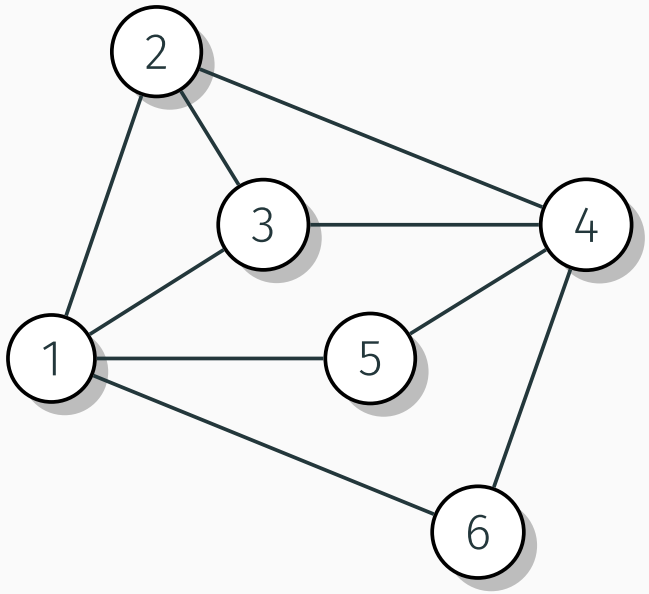
Hranu spojující vrchol se sebou samým nazýváme **smyčkou**.

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.

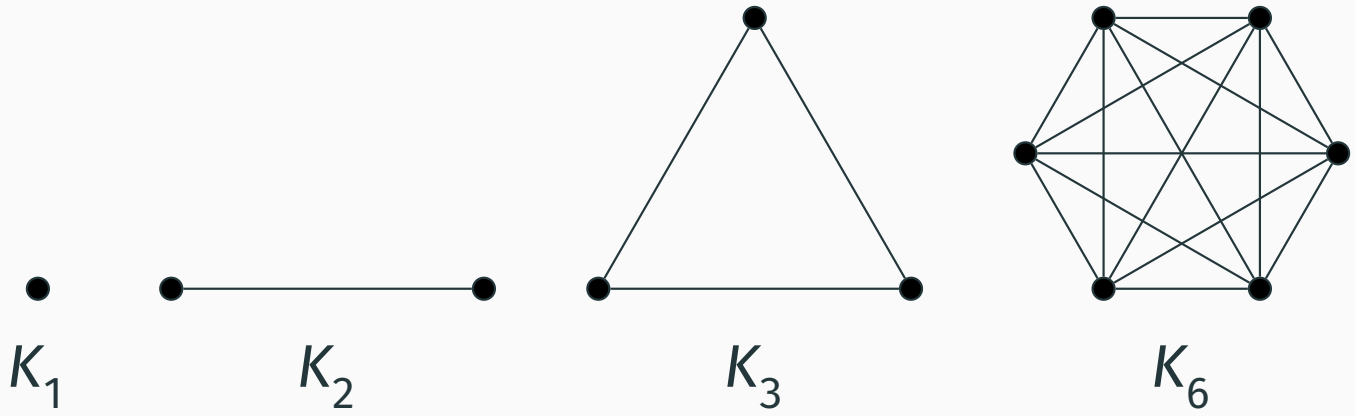
Maxmálního počtu hran v grafu docílíme tak, že každý z $|V|$ vrcholů spojíme hranou se všemi ostatními vrcholy, kterých je $|V| - 1$. Součin $|V|(|V| - 1)$ musíme vydělit dvěma, protože jsme každou hranu započítali dvakrát. □

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.

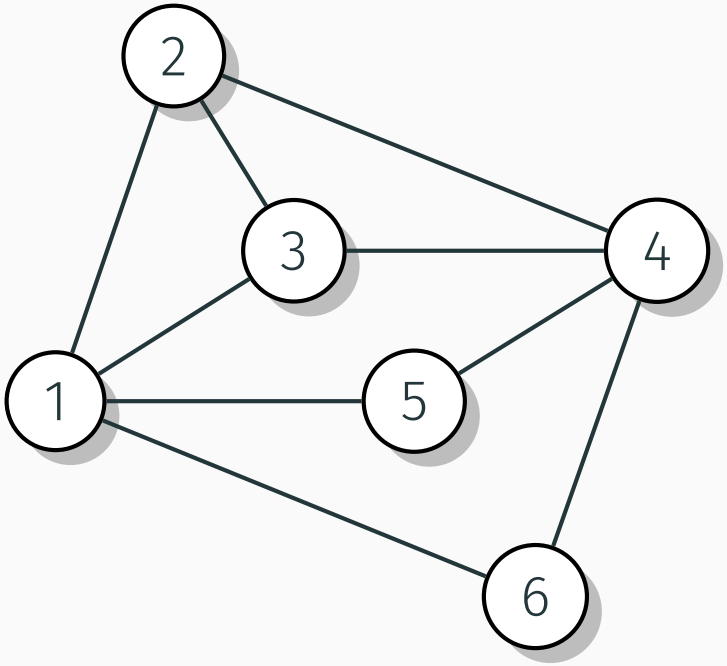
Podgraf (pokrač.)

Poznámky

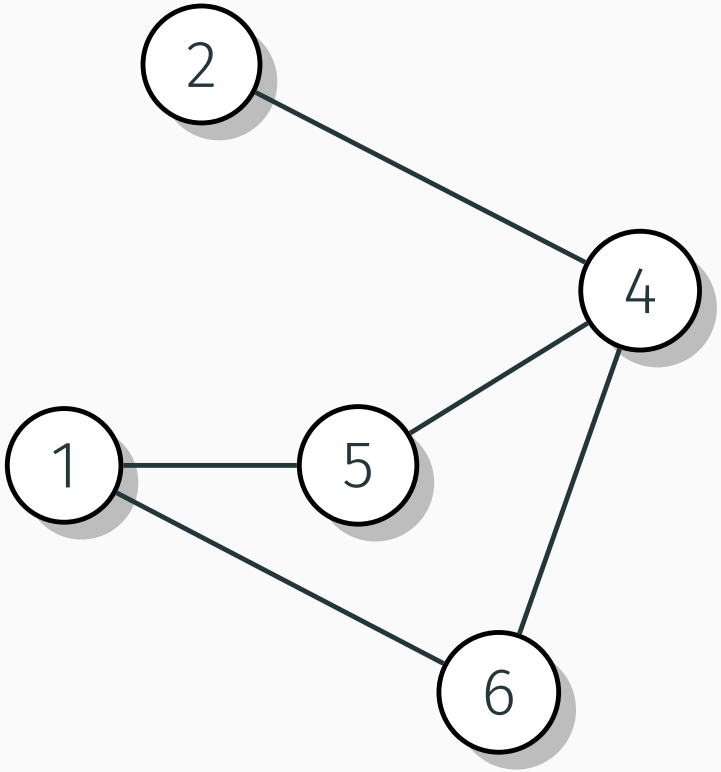
- Jinými slovy, podgraf vznikne vymazáním některých vrcholů původního grafu, všech hran do těchto vrcholů zasahujících a případně některých dalších hran.
- Termín podgraf se v teorii grafů používá jako jistá obdoba pojmu podmnožina.

Podgraf (pokrač.)

Příklad



Graf G

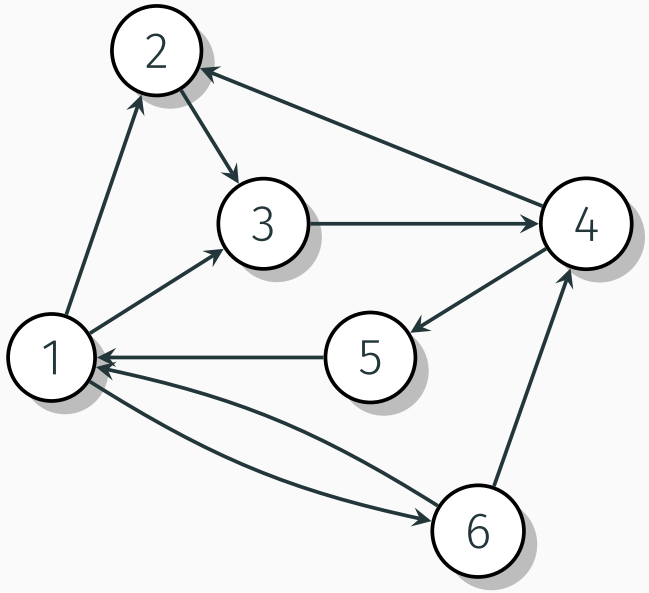


Podgraf H

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

$G = (V, E)$

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

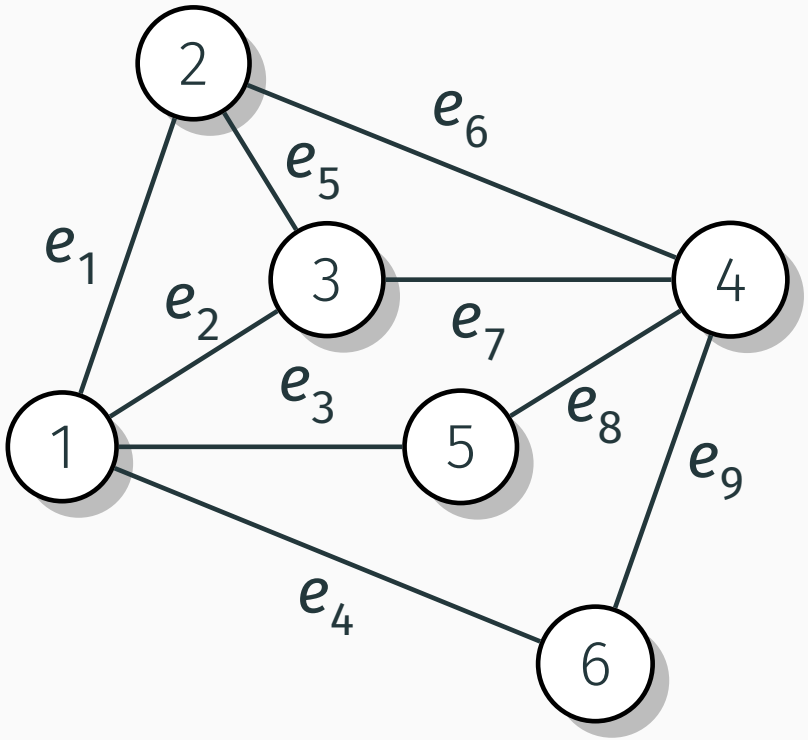
$E = \{(1, 2), (1, 3), (1, 6), (2, 3), (3, 4), (4, 2), (4, 5), (5, 1),$
$\{(6, 1), (6, 4)\}$

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.

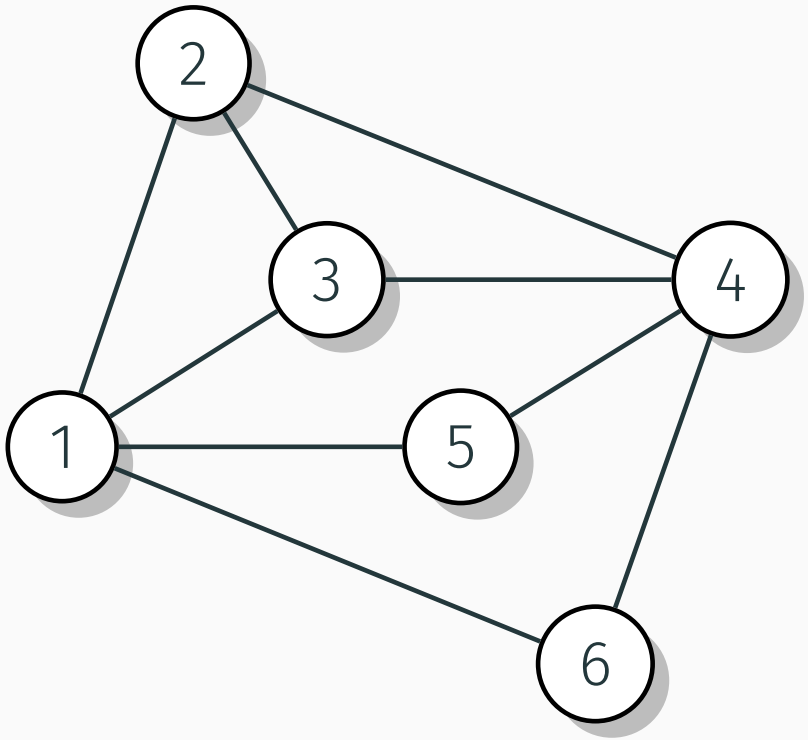


Matice incidence

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

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.

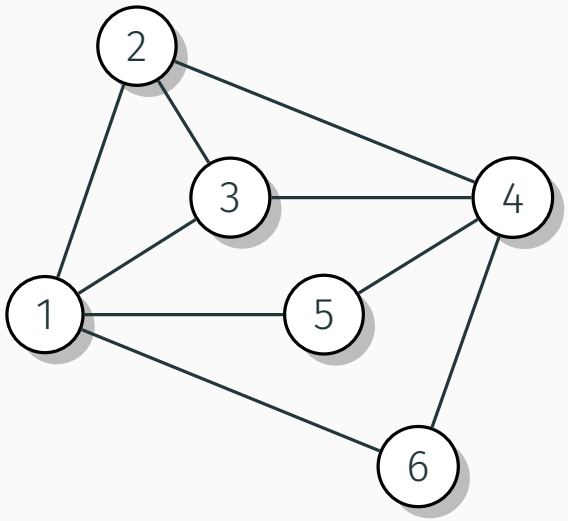


Matice sousednosti

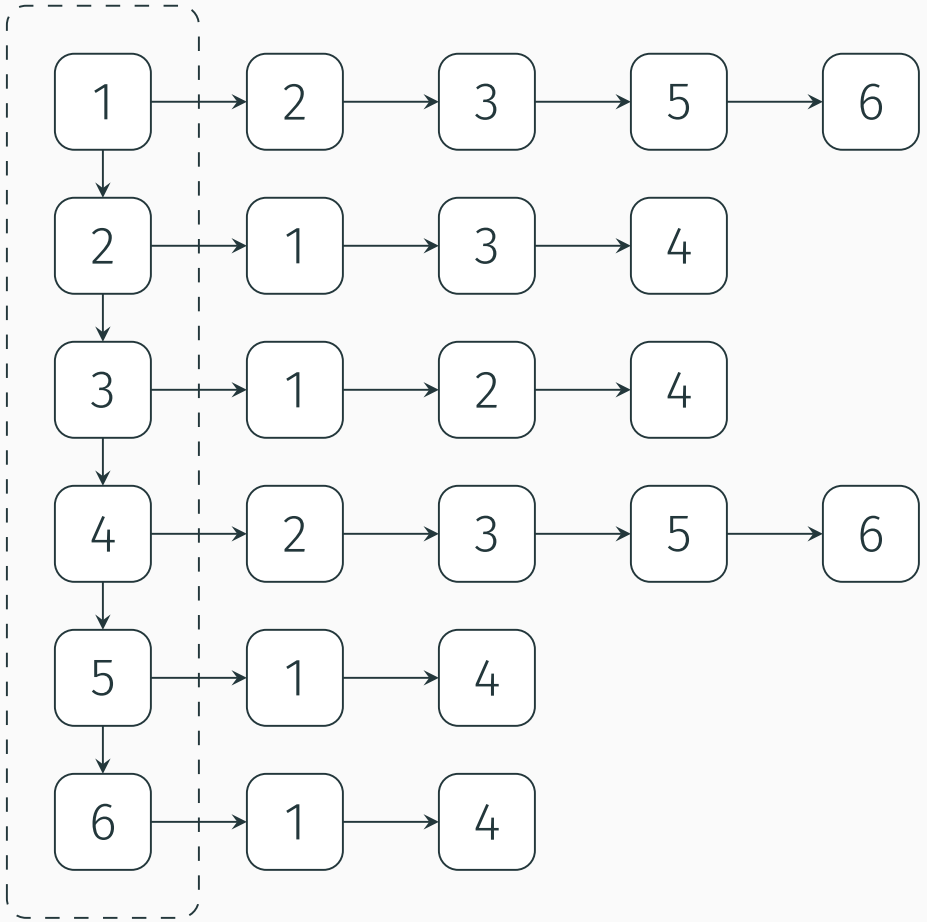
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Seznam sousedících vrcholů

Graf



Seznam sousedících vrcholů



Hustý vs. řídký graf a jeho reprezentace

Seznam sousedících vrcholů

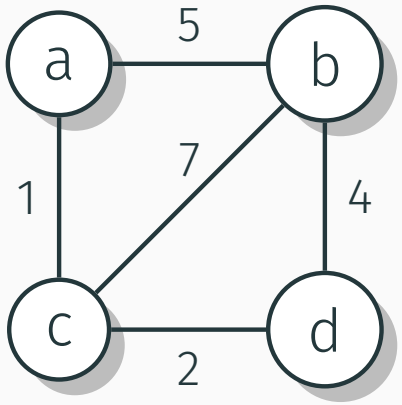
- ukazatele v seznamech zabírají paměť navíc,
- vhodný pro řídké grafy,
- pohodlnější změny struktury grafu (vložení smazání vrcholu, stejně tak hrany).

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.

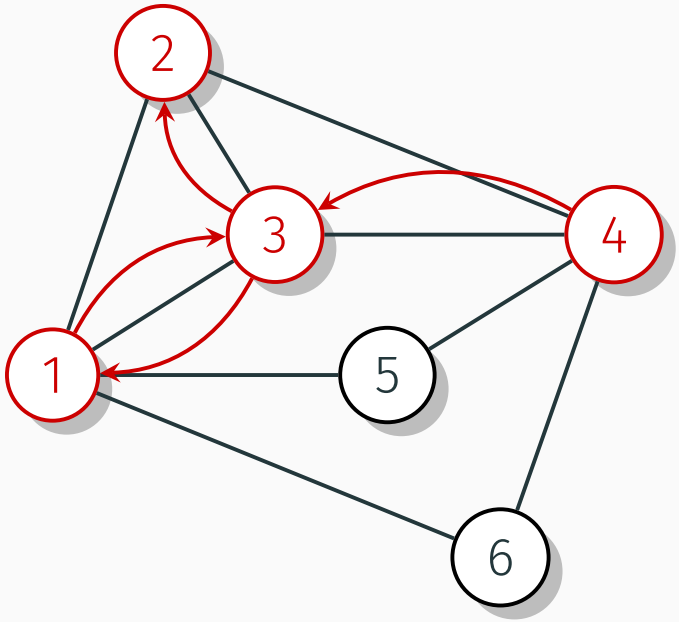


$$\begin{pmatrix} \infty & 5 & 1 & \infty \\ 5 & \infty & 7 & 4 \\ 1 & 7 & \infty & 2 \\ \infty & 4 & 2 & \infty \end{pmatrix}$$

Sled

Definice

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



Sled

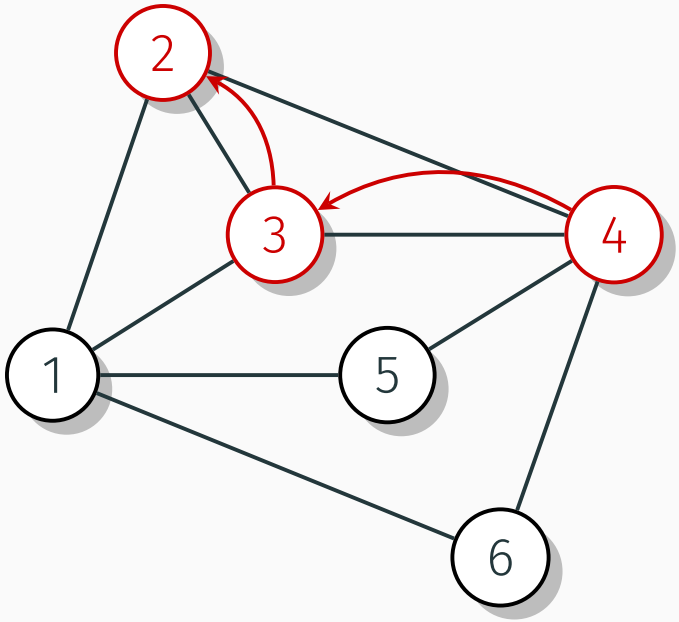
4 {4, 3} 3 {3, 1} 1 {1, 3} 3 {3, 2} 2

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

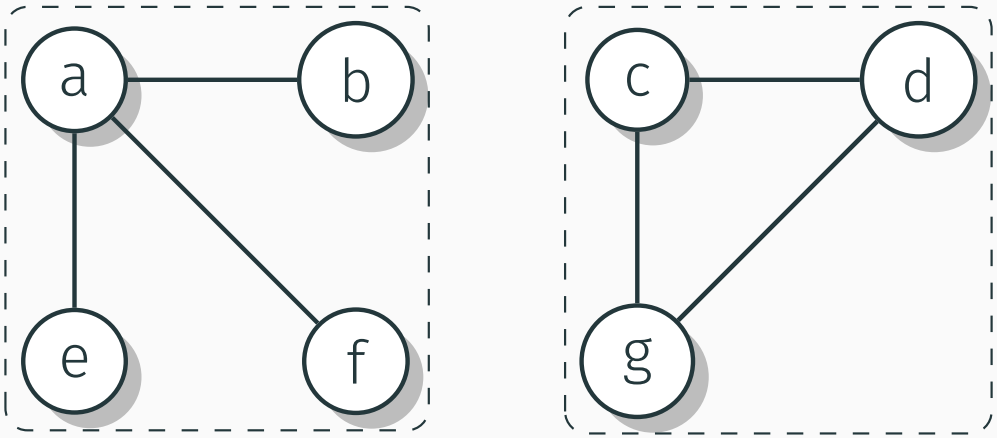
Graf se nazývá **souvislý**, jestliže mezi každými dvěma vrcholy existuje cesta.

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 $|E| \geq |V| - 1$.

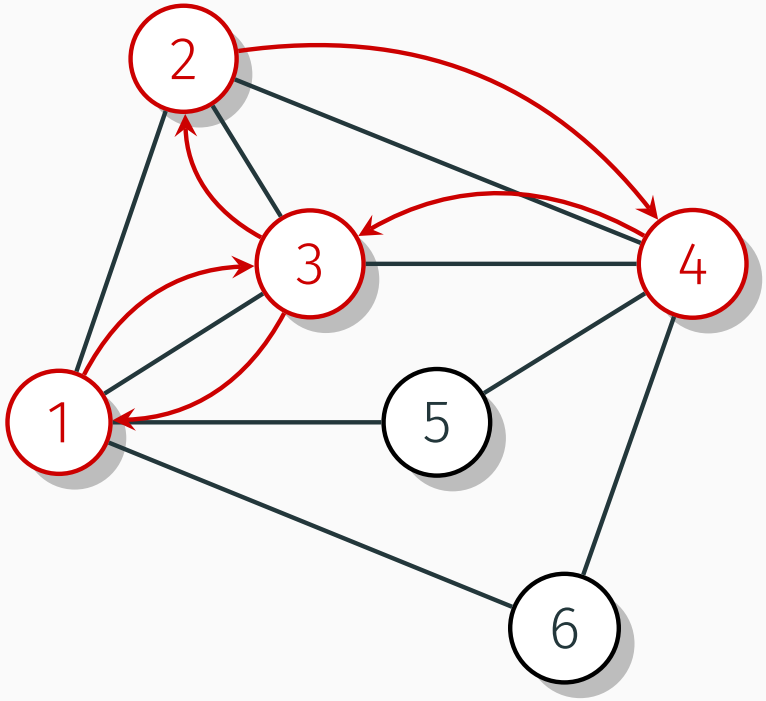
Důkaz.

Zřejmý. □

Uzavřený sled

Definice

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



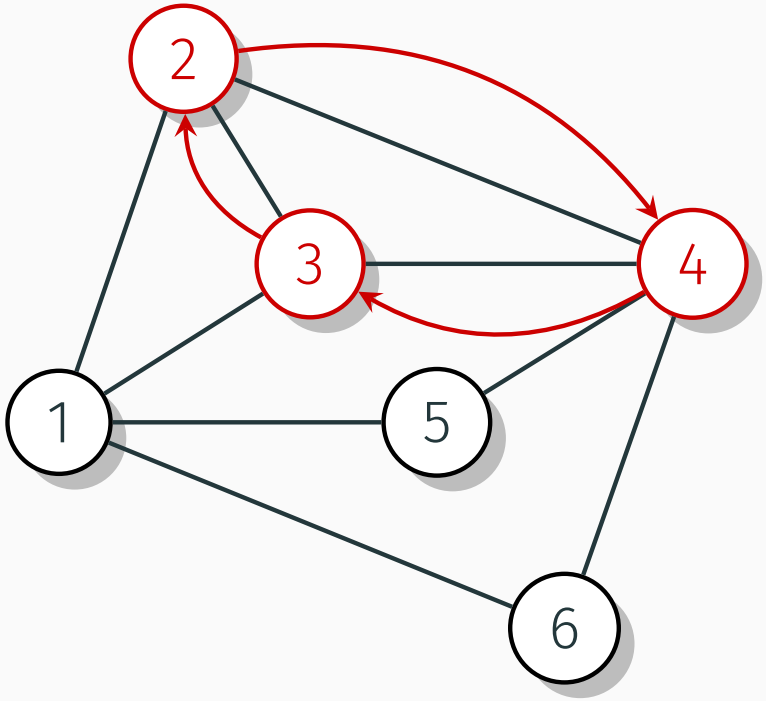
Uzavřený sled

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

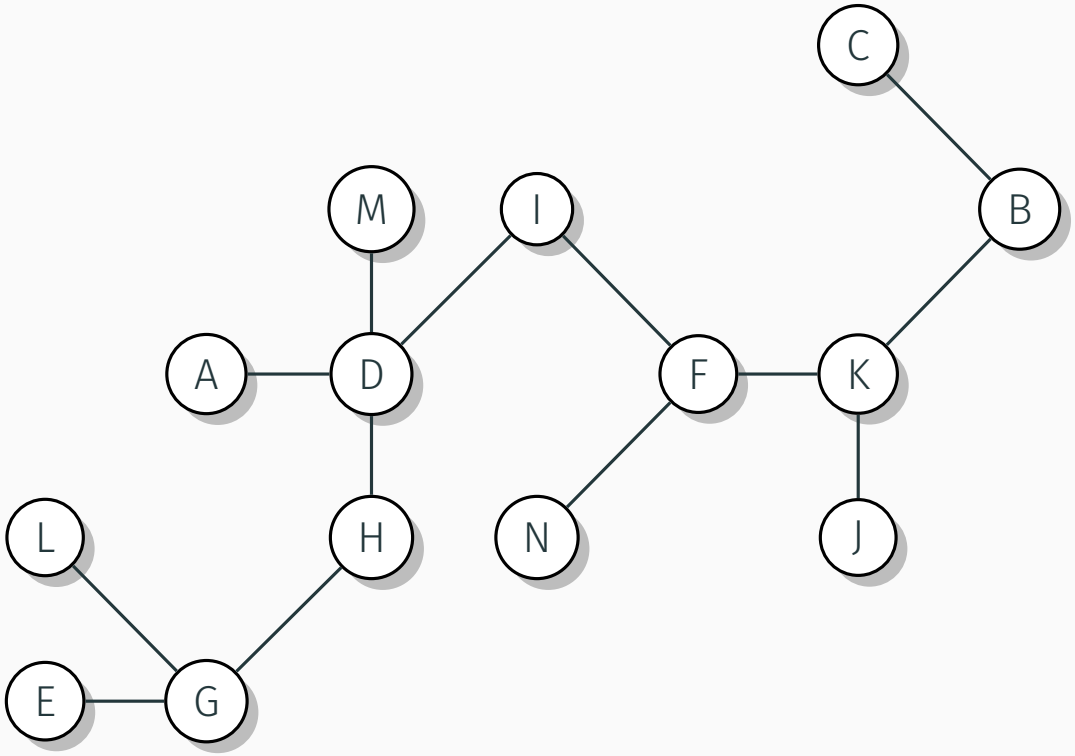
Definice

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

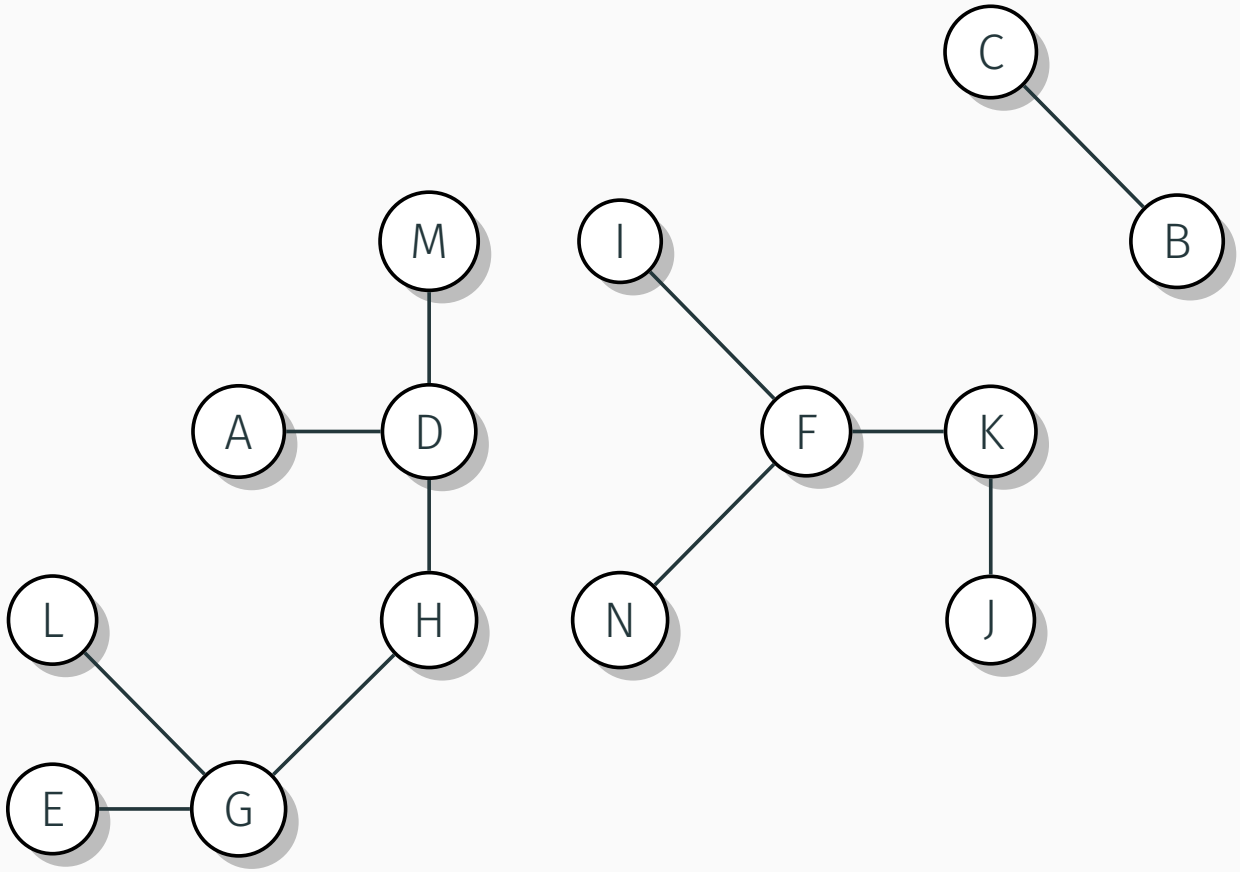
- V teorii grafů se objekty, které propojujeme hranami nazývají obvykle vrcholy (angl. vertex, vertices).
- Pokud mluvíme o stromech lze pro vrchol používat i výraz **uzel** (angl. node).
- Označení vrchol a uzel je rovnocenné, jde spíše o zvyklost.

Les

Definice

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

Každá souvislá komponenta lesa je volný strom



Vlastnosti volného stromu

Věta

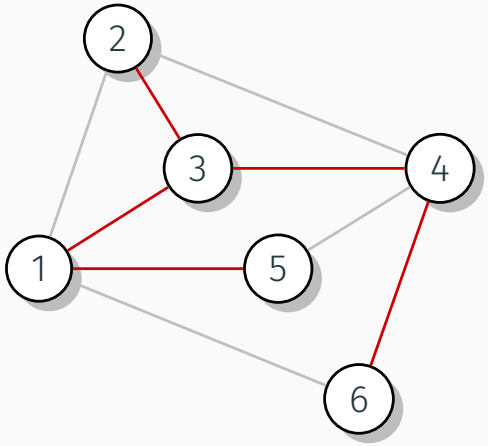
Nechť $G = (V, E)$ je neorientovaný graf, potom následující tvrzení jsou ekvivalentní:

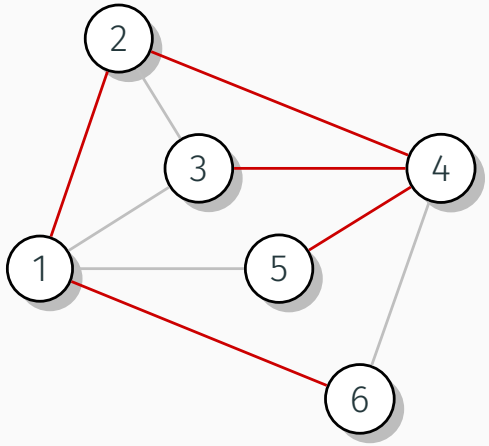
1. *G je volný strom.*
2. *Každé dva vrcholy v G jsou spojeny právě jednou cestou.*
3. *G je souvislý, ale pokud odebereme libovolnou hranu, získáme nesouvislý graf.*
4. *G je souvislý, a $|E| = |V| - 1$.*
5. *G je acyklický, a $|E| = |V| - 1$.*
6. *G je acyklický. Přidáním jediné hrany do množiny hran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

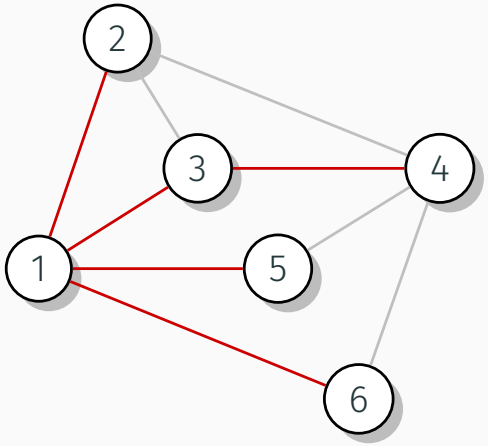
Kostra grafu (angl. Spanning tree)

Definice

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







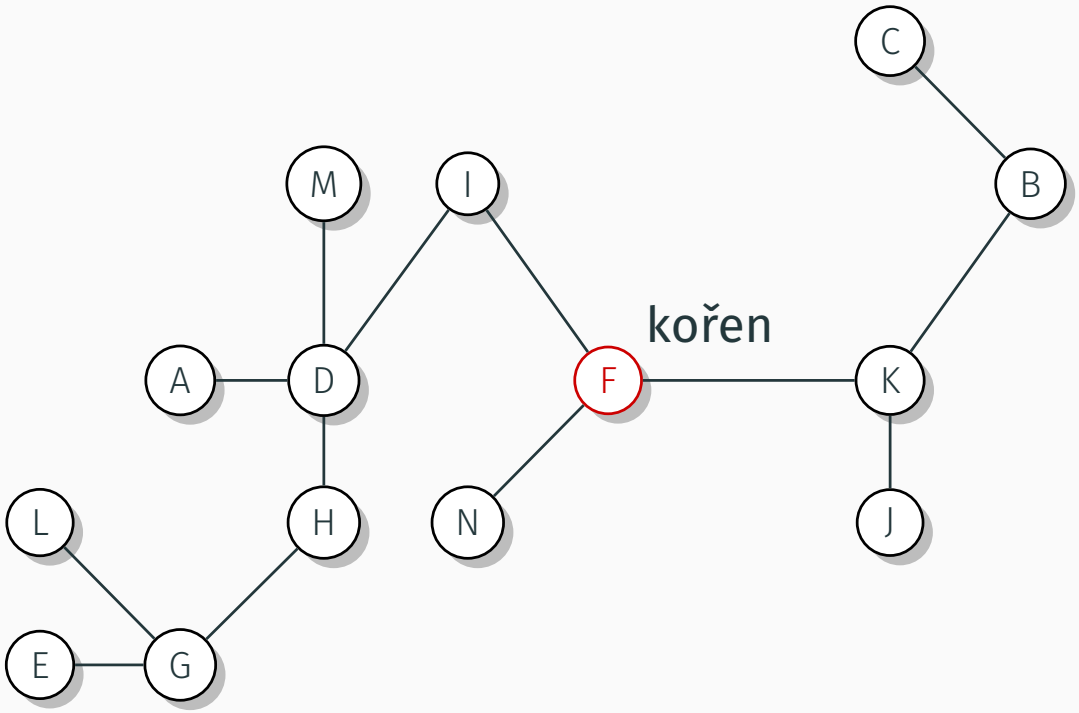
Poznámky

- 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.

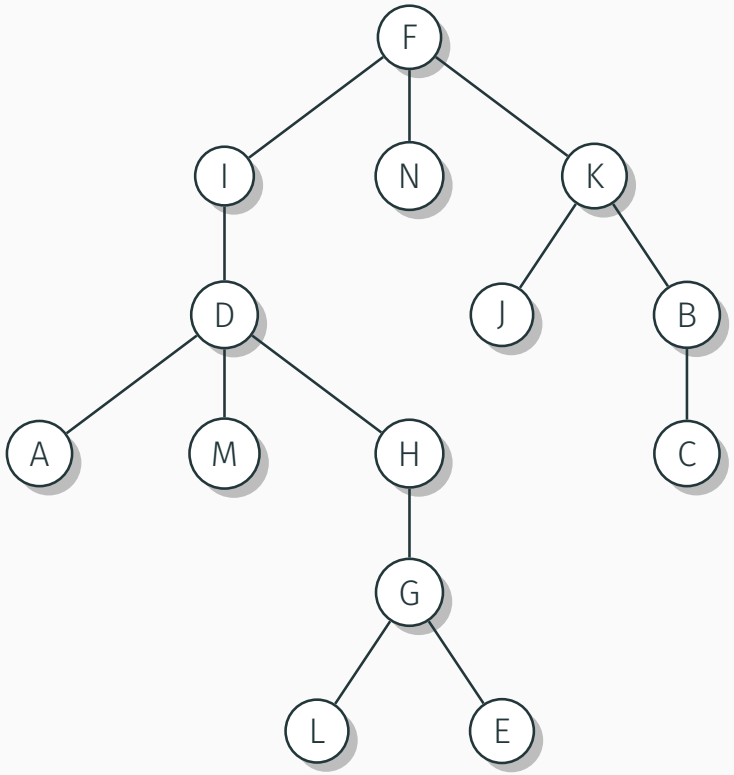


Poznámka

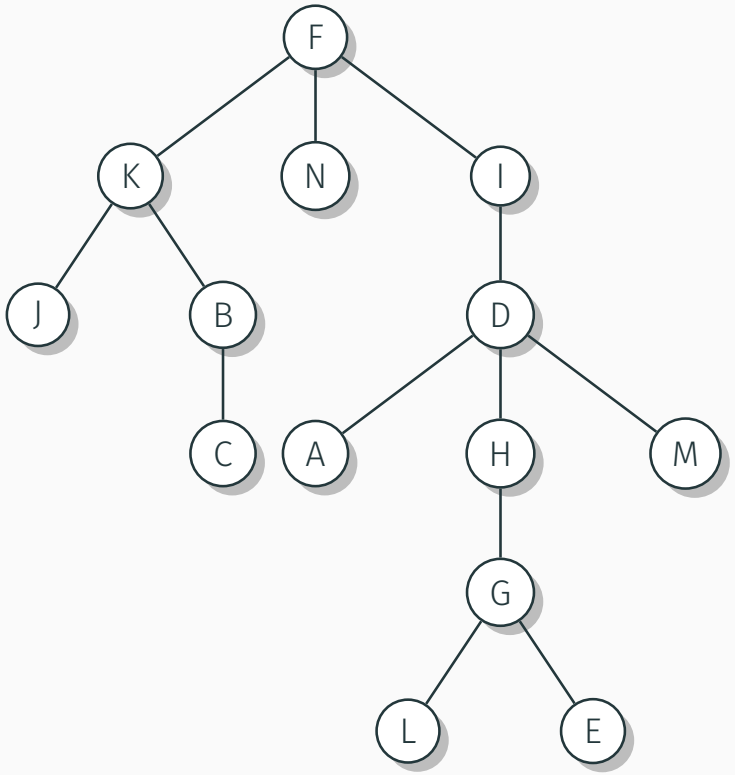
Někdy se používá také označení **zakořeněný strom**.

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.

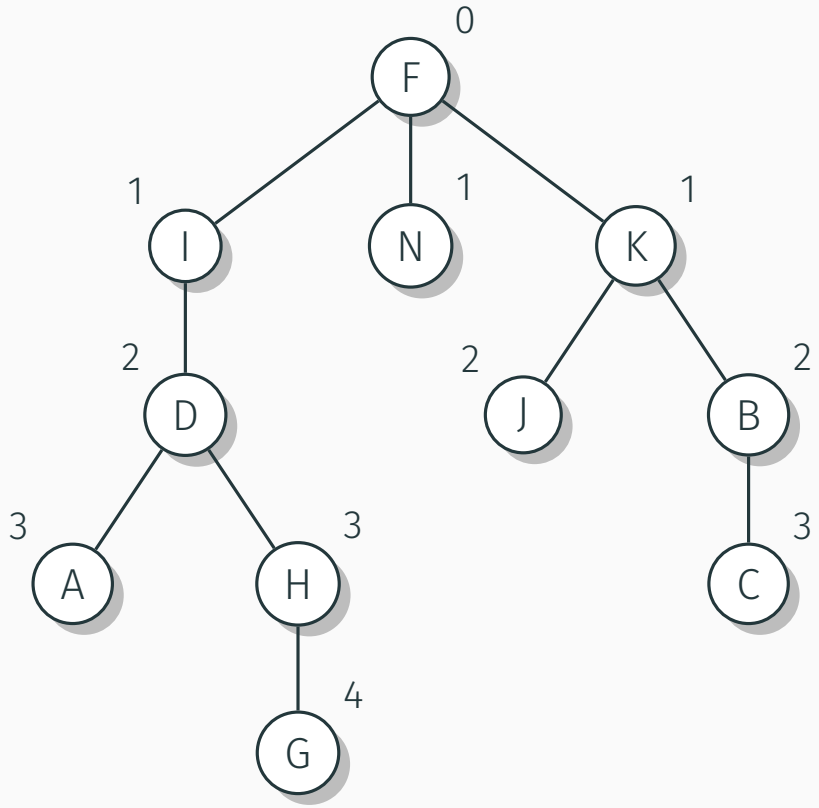
Hloubka vrcholu, výška stromu

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“.

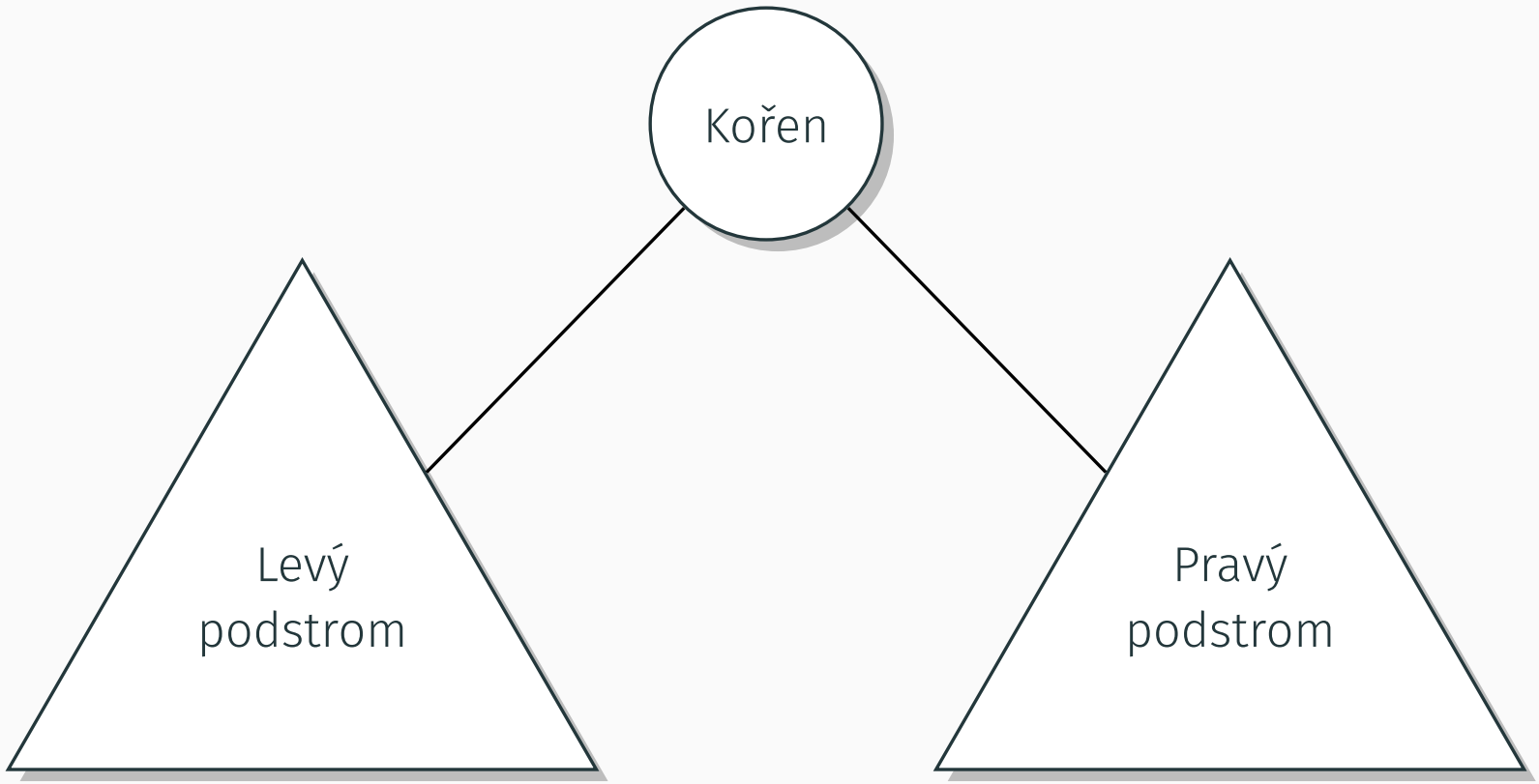
Binární strom

Definice

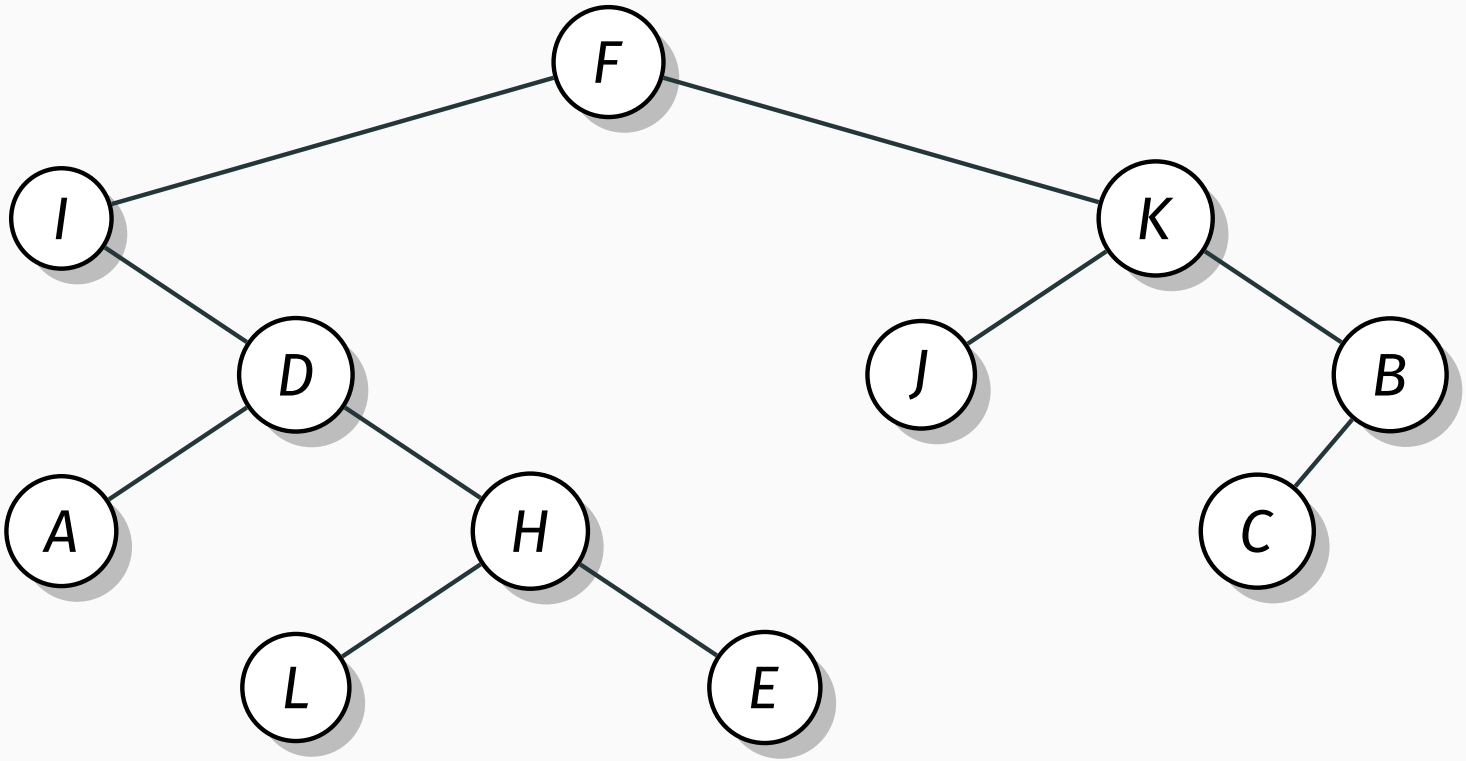
Binární strom je struktura definovaná nad konečnou množinou uzlů M, která:

- **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.

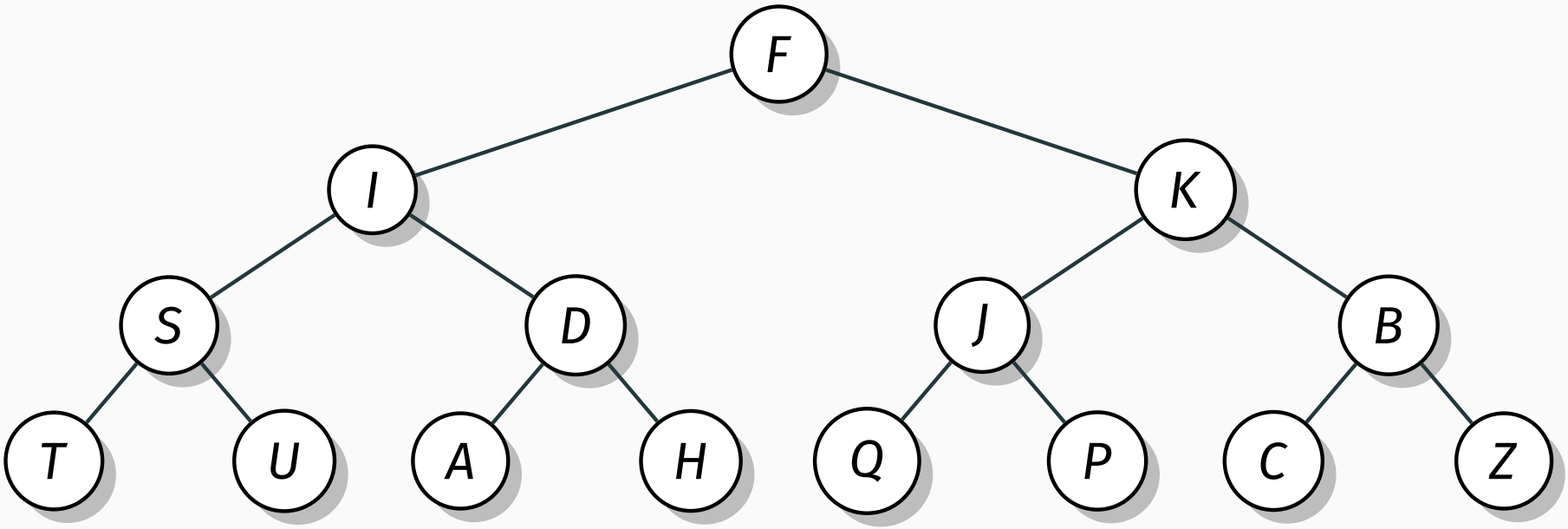
Binární strom – grafické znázornění rekurzivní definice



Binární strom, příklad



Úplný binární strom

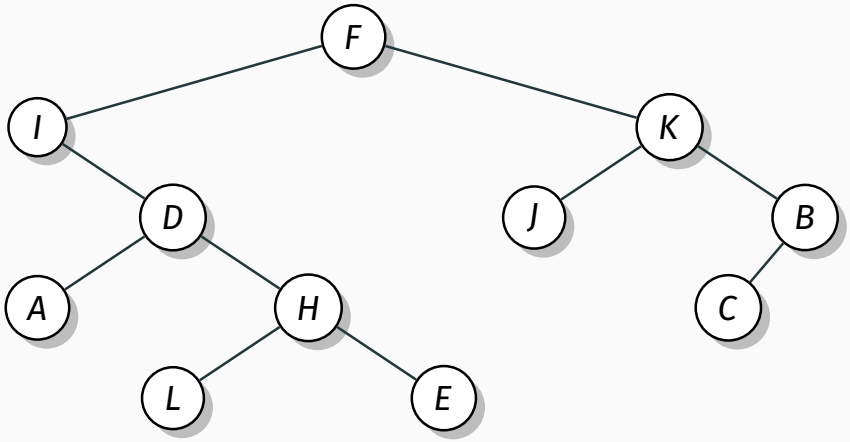


Úplný binární strom – každý vnitřní uzel má právě dva potomky.

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

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

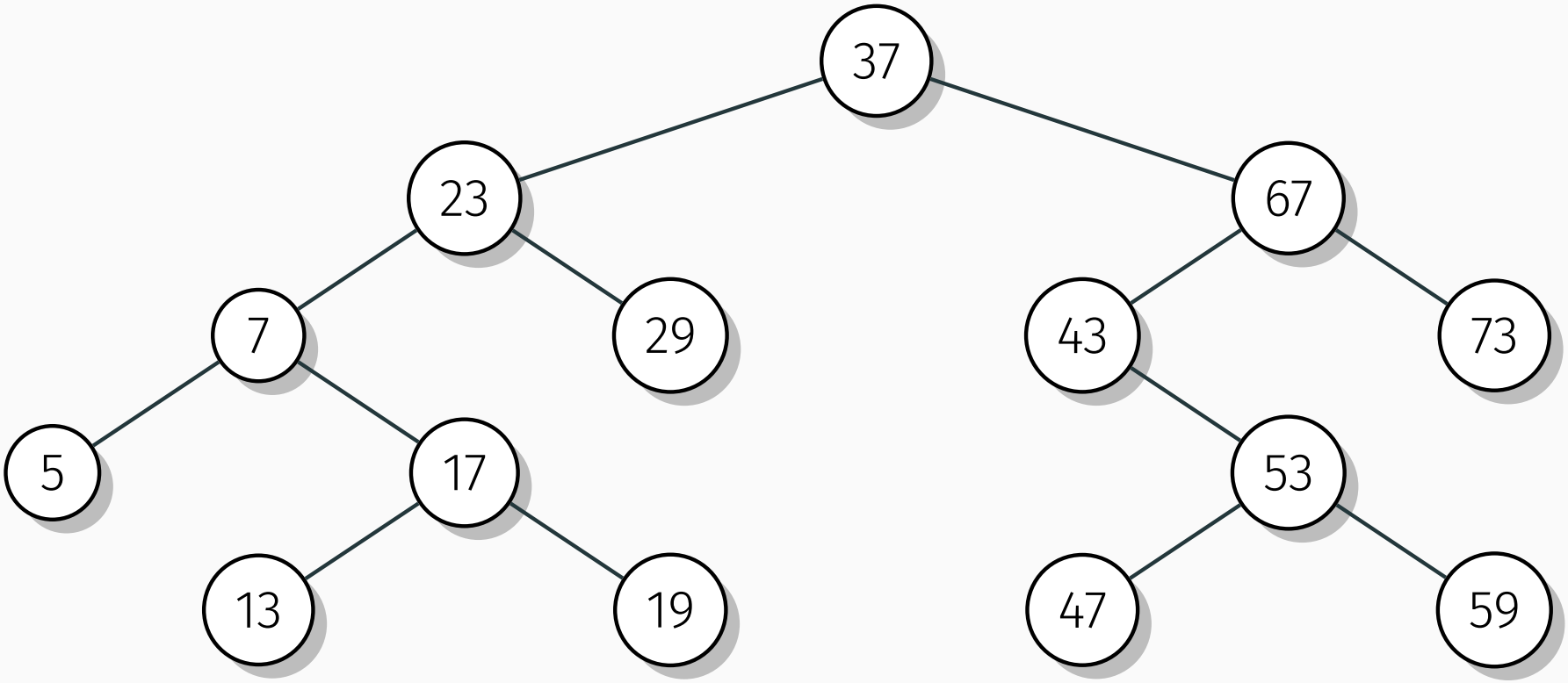
Binární strom, ve kterém pro všechny jeho uzly platí toto pravidlo nazýváme **binární vyhledávací strom** (angl. binary search tree).

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.

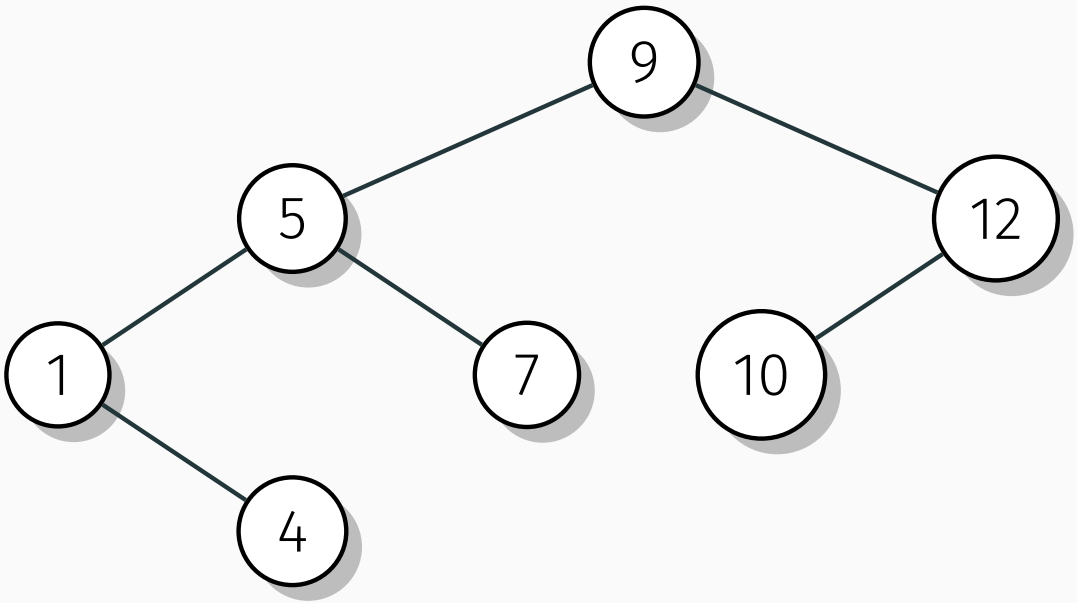
Pro výšku h binárního stromu s n uzly platí nerovnost

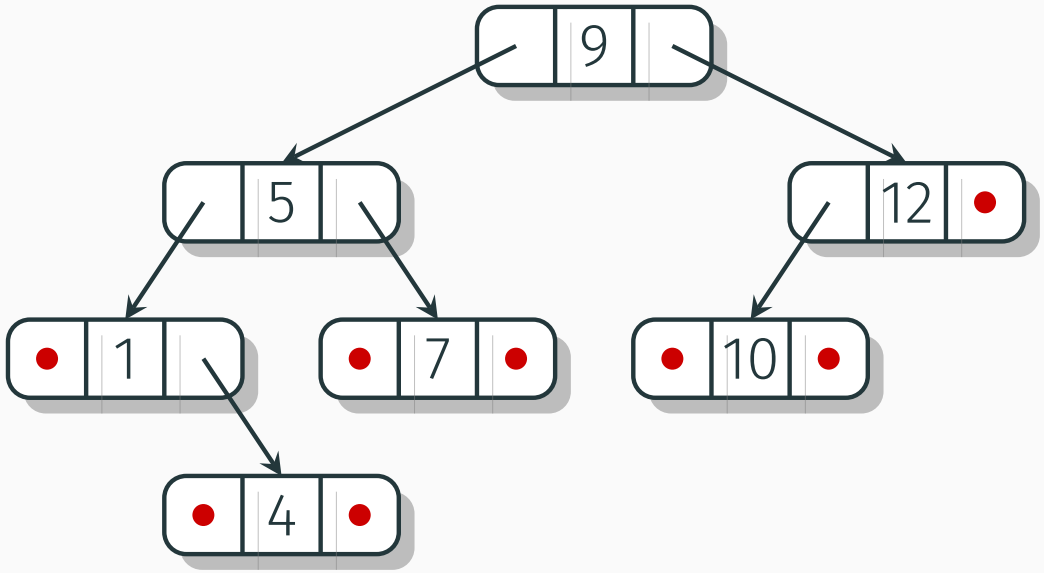
$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Binární vyhledávací strom – vkládání

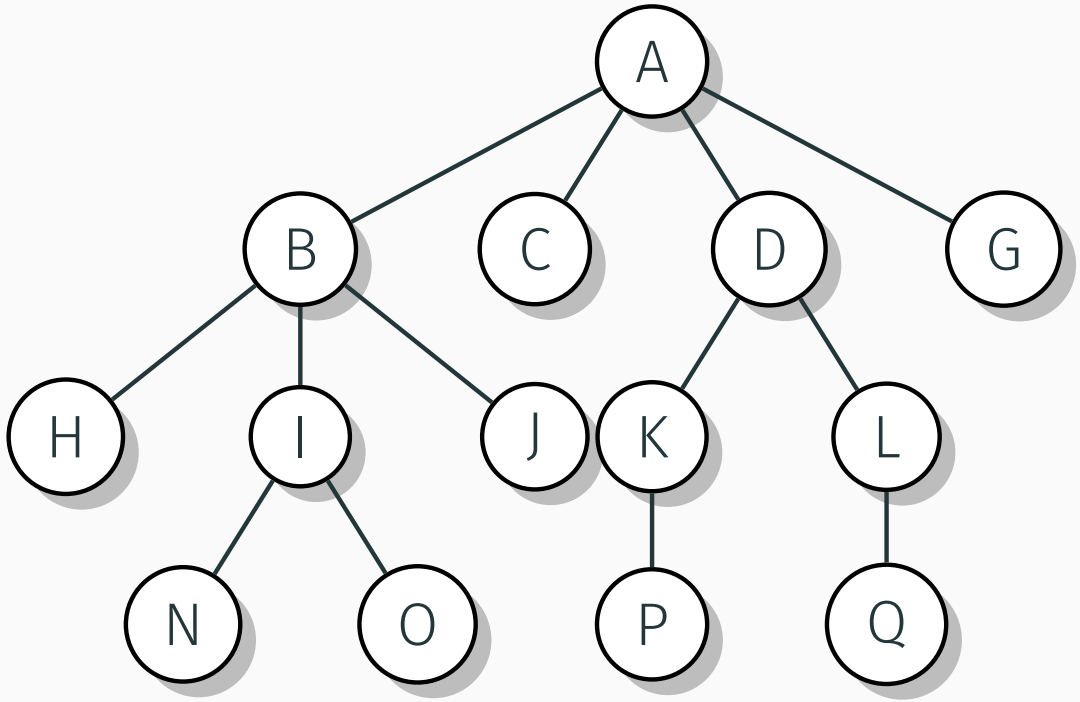
- Vložení klíče musí korespondovat s vyhledávacím algoritmem.
- Nejdříve se musíme pokusit vkládaný klíč ve stromu najít.
- Pokud jej nenajdeme, tak místo, kde jsme hledání neúspěšně zakončili odpovídá místu ve stromu, kde by tento klíč měl být.
- Toto plyne z jednoznačnosti cesty mezi kořenem a kterýmkoliv 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



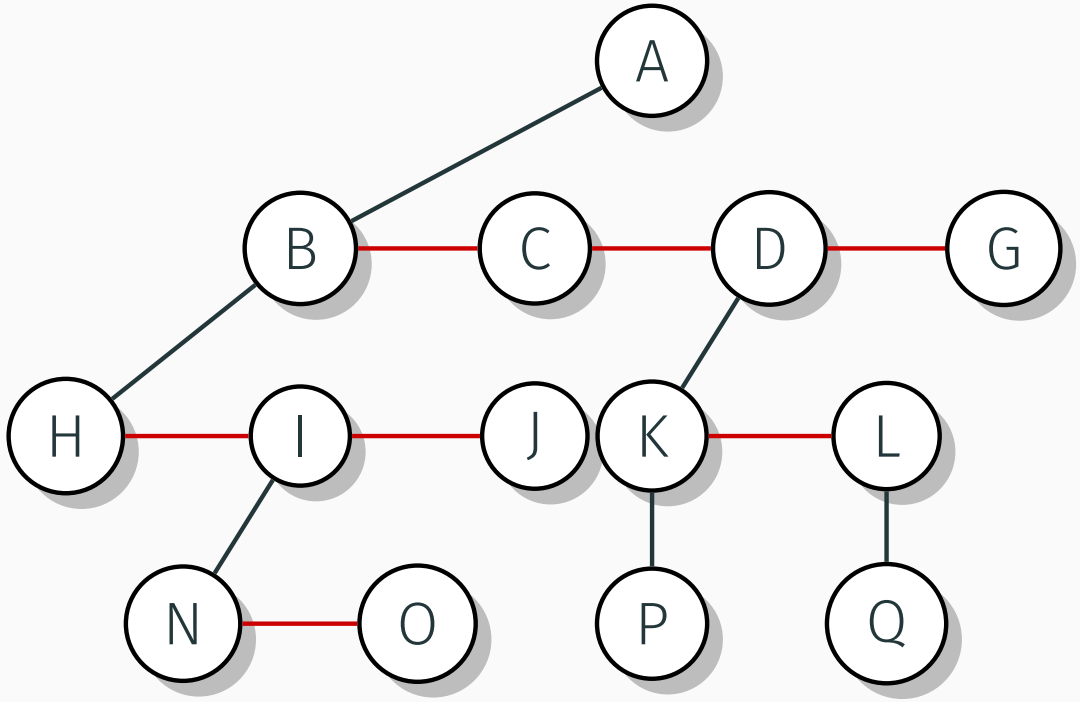


Reprezentace seřazeného stromu



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

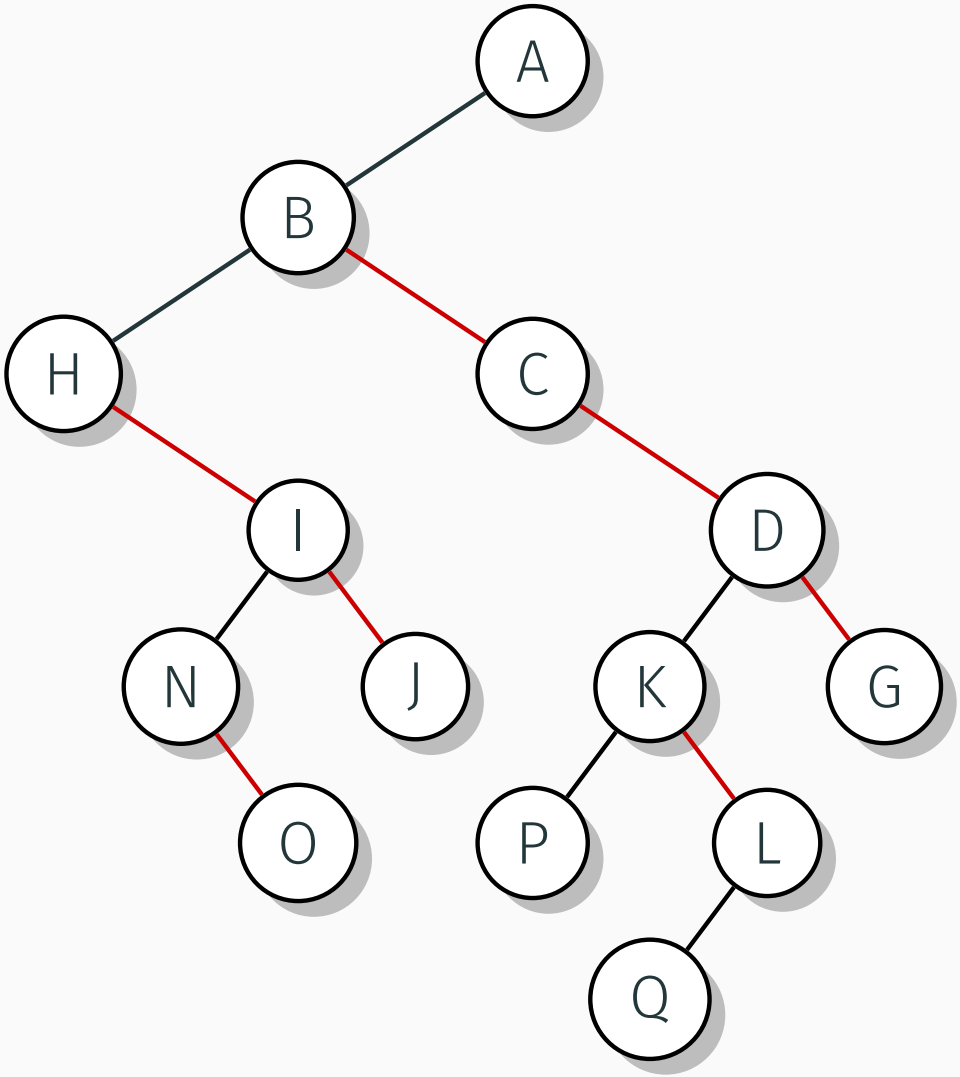
Reprezentace seřazeného stromu – first child – next sibling



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 $U = \{u_0, u_1, \ldots, u_{n-1}\}$, $|U| = n$
- 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

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

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

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

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

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Analýza složitosti algoritmů

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Analýza složitosti algoritmů

Základy analýzy složitosti algoritmů

Analýza algoritmů

Co analyzovat?

- správnost
- časová složitost
- prostorová složitost
- optimalita

Možné přístupy

- empirický a
- teoretický

Časová a prostorová složitost algoritmu

- **Časová složitost** – jak dlouho algoritmus bude pracovat.
- **Prostorová složitost** – kolik paměti bude algoritmus potřebovat **navíc** než je samotné uložení dat.
- Dříve byly oba zdroje kritické.
- Díky pokroku ve výpočetní technice je paměti relativně dost.
- Budeme zkoumat časovou složitost – tady lze dosáhnout významného pokroku.
- Ukazuje se, že prostorovou složitost lze zkoumat stejným aparátem jako časovou

Měření velikosti vstupu

- Triviální pozorování – větší data algoritmus obvykle zpracovává déle.
- Zavedeme parametr n označující velikost vstupních dat, který představuje například:
  - hledání v seznamu, poli – délka pole
  - vyhodnocení polynomu $p(x) = a_n x^n + \cdots + a_1 x + a_0$ v bodě x – stupeň polynomu
  - násobení matic typu $n \times n$ – rozměr matice. Skutečný počet čísel na vstupu je ale n^2, což je ale pořád závislé na n
  - kontrola pravopisu – počet znaků nebo počet slov, podle toho s čím algoritmus pracuje

Měření velikosti vstupu (pokrač.)

- test prvočíselnosti – vstupem je vždy jedno číslo (?!) a, doba běhu závisí na velikosti čísla (srovnej test 2^3 a 2^{64}), velikostí vstupu bude počet bitů nutných k zápisu čísla

$$n = \lfloor \log_2 a \rfloor + 1 \tag{2}$$

- grafové úlohy – počet vrcholů a/nebo počet hran – zde už jsou dva parametry

Empirické měření složitosti

- Zadáme vhodná (?) vstupní data a změříme dobu běhu programu v obvyklých jednotkách času.
- Nevýhody:
  - Závislost na konkrétním HW, způsobu implementace, kompilátoru.
  - Chceme měřit složitost algoritmů – nemáme prostředky pro zachycení výše uvedených vlivů.
  - Vývoj HW – znamená to, že se algoritmy zrychlují? Ne, ty zůstávají stejné.
  - Počet operací provedených programem lze obtížné zjistit.
  - Chceme se obejít bez implementace – zkoumáme přece algoritmy.

Časová složitost algoritmu

Časovou složitost algoritmu budeme vyjadřovat (měřit) **počtem vykonaných základních operací** vzhledem k (jako funkci) **velikosti vstupu** n:

$$T(n) \approx c_{op} C(n),$$

kde

- n je velikost vstupu,
- $T(n)$ je doba běhu algoritmu,
- c_{op} je doba vykonání jedné základní operace a
- $C(n)$ je počet základních operací.

Základní operace

Typické operace pro daný algoritmus, které významně přispívají ke celkové „době běhu“ algoritmu.

Problém	Měřítko vstupu	Základní operace
Hledání prvku v seznamu	Počet prvků v seznamu	Porovnání prvků
Násobení matic	Rozměry matic	Aritmetické operace (násobení)
Test prvočíselnosti	Počet bitů čísla	Dělení čísel
Grafové úlohy	Počet vrcholů a / nebo hran	Zpracování vrcholu či průchod hranou

Řádový růst složitosti

Ke vztahu

$$T(n) \approx c_{op} C(n),$$

je potřeba ale přistupovat „s rezervou“, protože

1. $C(n)$ nebere v úvahu vliv jiných operací než základních a
2. c_{op} nelze spolehlivě zjistit.

Vztah chápeme jako **rozumný odhad doby běhu algoritmu**, mimo extrémně malých n.

Řádový růst složitosti (pokrač.)

Problém: Kolikrát rychleji poběží můj algoritmus na počítači, který je **10×** rychlejší než můj současný počítač?

Řešení:

Samozřejmě **10×**, c_{op} je desetinové.

Řádový růst složitosti (pokrač.)

Problém: Kolikrát déle poběží můj algoritmus pro dvojnásobně velký vstup, když $C(n) = \frac{1}{2}n(n-1)$?

Řešení: Aproximujeme shora počet operací $C(n)$

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n < \frac{1}{2}n^2$$

a odtud

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = \frac{4n^2}{n^2} = 4$$

Zanedbáno	Podstatné
Lineární člen $\frac{1}{2}n$	Kvadratický růst složitosti – mluvíme o **řádovém růstu**.

Řádový růst složitosti (pokrač.)

Tabulka ukazuje, jak rychle, či pomalu, rostou hodnoty vybraných funkcí pro různá n.

	Hodnota funkce tj. počet operací $C(n)$						
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	$3,3$	10^1	$3,3 \cdot 10^1$	10^2	10^3	10^3	$3,6 \cdot 10^6$
10^2	$6,6$	10^2	$6,6 \cdot 10^2$	10^4	10^6	$1,3 \cdot 10^{30}$	$9,3 \cdot 10^{157}$
10^3	10	10^3	$1,0 \cdot 10^4$	10^6	10^9	n/a	n/a
10^4	13	10^4	$1,3 \cdot 10^5$	10^8	10^{12}	n/a	n/a
10^5	17	10^5	$1,7 \cdot 10^6$	10^{10}	10^{15}	n/a	n/a
10^6	20	10^6	$2,0 \cdot 10^7$	10^{12}	10^{18}	n/a	n/a

Pro malé n je rozdíl mezi hodnotami funkcí vcelku nezajímavý, ale pro zvětšující se n může být rozdíl propastný.

Řádový růst složitosti (pokrač.)

Poznámky

- Základ logaritmu není podstatný: $\log_a n = \log_a b \cdot \log_b n$.
- Počítači s rychlostí 10^{12} (tisíc miliard) operací za sekundu trvá provedení $2^{100} \approx 1,3 \cdot 10^{30}$ operací cca 40 miliard let. Stáří Země je cca 4,4 miliard let.
- O provedení **100!** operací nebudeme ani uvažovat...

Algoritmy s exponenciální nebo faktoriálovou řádovou složitostí jsou použitelné jen pro velice malé velikosti vstupu!

Řádový růst složitosti (pokrač.)

Problém: Kolikrát déle poběží můj algoritmus pro dvojnásobně velký vstup, pro algoritmy s různým řádovým růstem?

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
$2n$	$+1$	$2\times$	$\approx 2\times$	$4\times$	$8\times$	$(\dots)^2$	n/a

protože

$$\log_2(2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$$

$$2^{2n} = (2^n)^2$$

Analýza složitosti algoritmů

Nejhorší, nejlepší a průměrný případ

Nejhorší, nejlepší a průměrný případ

- Počet základních operací udáváme jako funkci s jedním parametrem n, velikostí vstupu.
- Některé algoritmy můžou mít i pro stejné n různé počty zákl. operací, například algoritmus lineárního vyhledávání.

Vstup : Pole $A[0 \dots n-1]$ a hledaný prvek x
Výstup: Index prvního výskytu prvku x v poli A, jinak -1

```
1 for i ← 0 to n – 1 do
2 |   if A[i] = x then
3 |   |   return i;
4 |   end
5 end
6 return -1;
```

Nejhorší, nejlepší a průměrný případ (pokrač.)

Významné počty základních operací:

- $C_{worst}(n)$ – nejhorší případ, nejvyšší počet operací
- $C_{best}(n)$ – nejlepší případ, nejnižší počet operací
- $C_{avg}(n)$ – průměrný případ, průměrný počet operací.

Nejhorší případ $C_{worst}(n)$

- Analyzujeme algoritmus a hledáme vstup velikosti n pro který nastane nejvyšší možný počet operací.
- Nejhorší případ poskytuje horní mez složitosti, všechny ostatní případy jsou buď stejné nebo lepší.
- Nízký počet operací v nejhorším případě – pozitivní zpráva.

Příklad

Lineární vyhledávání: prvek x v poli A není nebo je nalezen až na konci, tedy $C_{worst}(n) = n$.

Nejlepší případ $C_{best}(n)$

- Obecně hledáme vstup velikosti n, pro který algoritmus vykoná nejmenší počet operací.
- Většinově nejlepší případ není tak důležitý jako nejhorší případ.
- Vstupy „podobné“, „blízké“ nejlepšímu. Třídění téměř setříděných posloupností.
- Nejlepší případ s „děsivým“ počtem operací – obecně špatná zpráva a „konečná“ pro algoritmus. Ale pro šifrovací algoritmus je „děsivý“ počet operací kryptoanalýzy i nejlepším případě nezbytný.

Příklad

Lineární vyhledávání: prvek x je prvním prvkem v poli A, $C_{best}(n) = 1$.

Průměrný případ $C_{avg}(n)$

- Počet operací v průměrném, „typickém“, „náhodném“ případě (nejlepší a nejhorší případy jsou extrémy).
- Nejedná se o průměr nejlepšího a nejhoršího případu!
- Musíme brát v úvahu pravděpodobnosti jednotlivých možných vstupů velikosti n.
- Analýza průměrného případu je tudíž komplikovanější než předchozích dvou.
- Existují algoritmy, kde se nejhorší a průměrný počet operací značně liší, např. QuickSort.

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání

Předpoklady

1. pravděpodobnost úspěšného vyhledání p, kde $0 \leq p \leq 1$
2. pravděpodobnost nalezení na všech pozicích v poli je shodná a je rovna $\frac{p}{n}$

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání (pokrač.)

Úspěšné vyhledání

- nalezení na první pozici – jedno porovnání s pravděpodobností $\frac{p}{n}$,
- nalezení na druhé pozice – dvě porovnání s pravděpodobností $\frac{p}{n}$, a tak dále, tedy

$$1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}$$

Neúspěšné vyhledání

- pravděpodobnost neúspěchu je $1 - p$ a provedeme n porovnání, tj. $n(1 - p)$

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání (pokrač.)

Odtud

$$\begin{aligned} C_{avg}(n) &= \left(1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}\right) + n(1-p) \\ &= \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1-p) \\ &= \frac{p}{n}\left[\frac{1}{2}n(n+1)\right] + n(1-p) \\ &= \frac{1}{2}p(n+1) + n(1-p) \end{aligned}$$

Rozbor

- vždy úspěšné hledání, $p = 1$ a tedy $C_{avg}(n) = \frac{1}{2}(n+1)$
- neúspěšné hledání, $p = 0$ a tedy $C_{avg}(n) = n$

Amortizovaná složitost

- Nezkoumáme jeden, izolovaný, běh algoritmu, ale zkoumáme „sadu“ běhů s různými vstupy stejné velikosti.
- Zajímá nás celkový počet operací za sadu.
- Počet operací pro jeden vstup ze sady může být sice vysoký, ale je vyvážen, „amortizován“ výrazně menším počtem operací pro další vstupy ze sady.
- Například jeden ze vstupů způsobí rozsáhlou změnu v datové struktuře a díky tomu zpracování dalších vstupů proběhne snadněji.
- V průmyslu je například nákup drahého stroje amortizován levnější výrobou výrobku.

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.1, strany 42 – 51
- Kniha [3], kapitola 2.2, strany 25 – 34 (částečně)

Analýza složitosti algoritmů

Asymptotické notace složitosti

O-notace

Definice

Mějme funkce $t(n)$ a $g(n)$, kde $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. Říkáme, že funkce $t(n)$ patří do $O(g(n))$, jestliže existuje kladná nenulová reálná konstanta c a přirozené číslo $n_0 > 1$ takové, že

$$t(n) \leq cg(n)$$

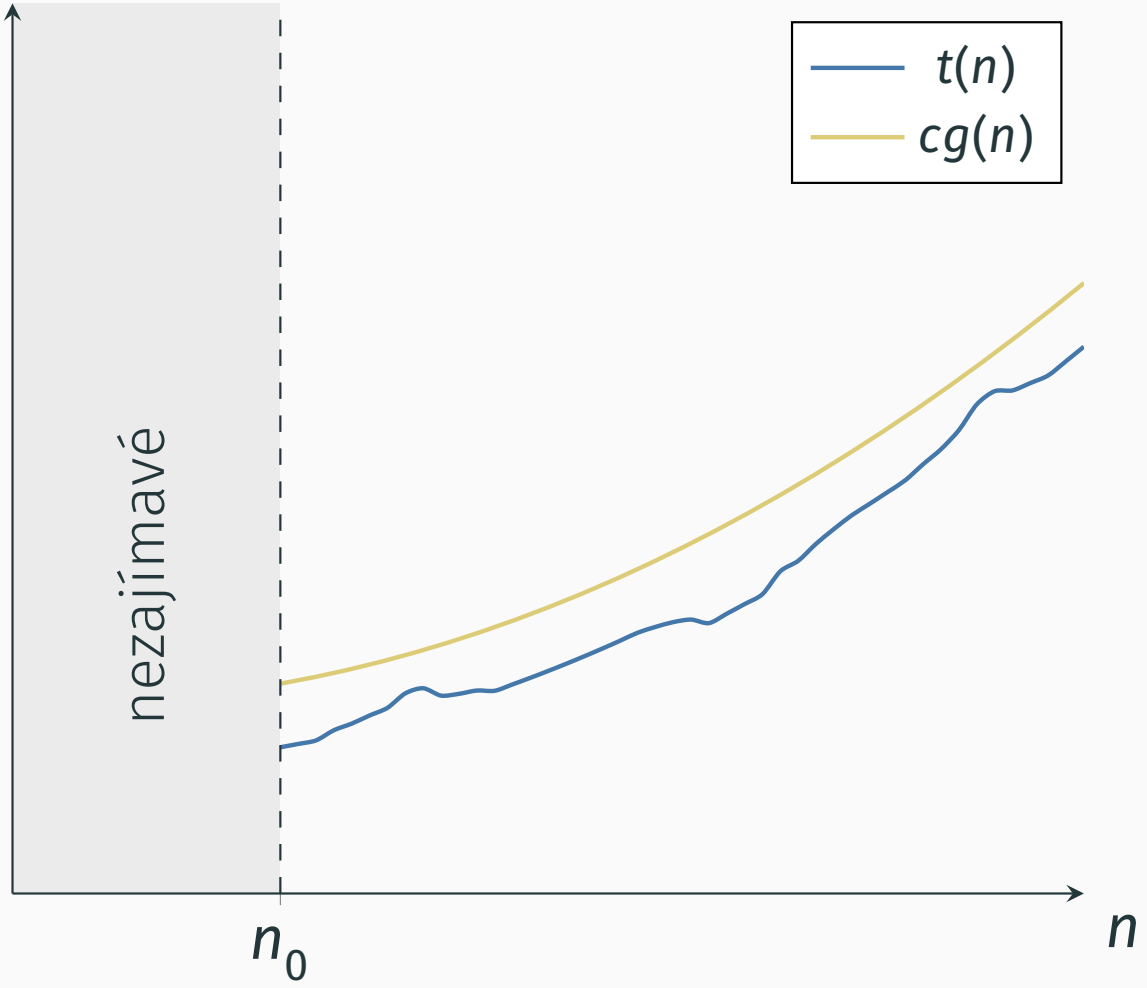
pro všechna $n \geq n_0$.

Poznámka

Místo „$t(n)$ patří do $O(g(n))$“ můžeme říkat, že „$t(n)$ je řádu $O(g(n))$“.

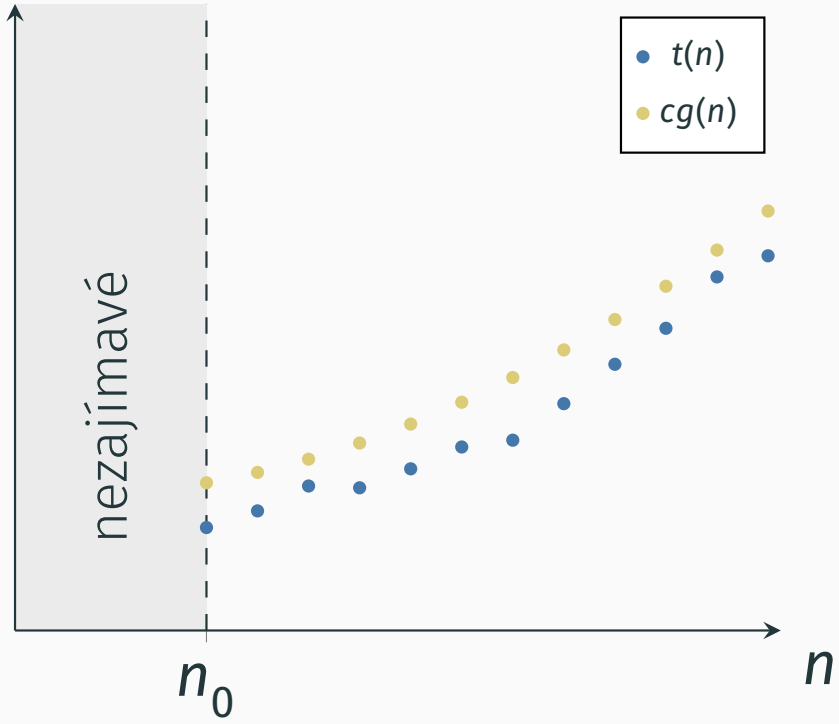
O-notace graficky

$$t(n) \in O(g(n))$$



O-notace – formálně správný graf

Formálně jsou definičním oborem i oborem hodnot funkcí $t(n)$ i $g(n)$ přirozená čísla ⇒ graf by měl být složen pouze z bodů, nikoliv křivek.



Proložíme-li body křivky ⇒ dostáváme spojité funkce ⇒ můžeme použít k výpočtům matematickou analýzu (limity, derivace atd.).

O-notace – příklad 1

Dokažte, že $3n + 7 \in O(n)$.

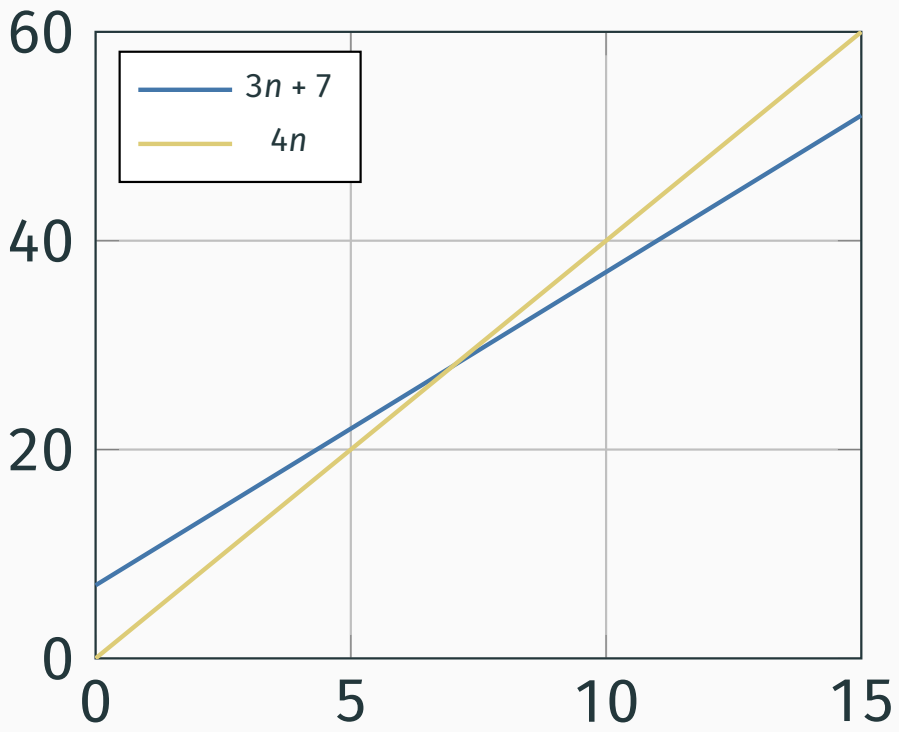
Řešení

1. Hledáme konstanty c a n_0 takové, aby platilo

$$3n + 7 \leq cn$$

pro všechna $n \geq n_0$.

2. Je zřejmé, že $c > 3$. Zvolíme-li např. $c = 4$, pak $n_0 = 7$.



O-notace – příklad 2

Dokažte, že $3n + 7 \in O(n^2)$

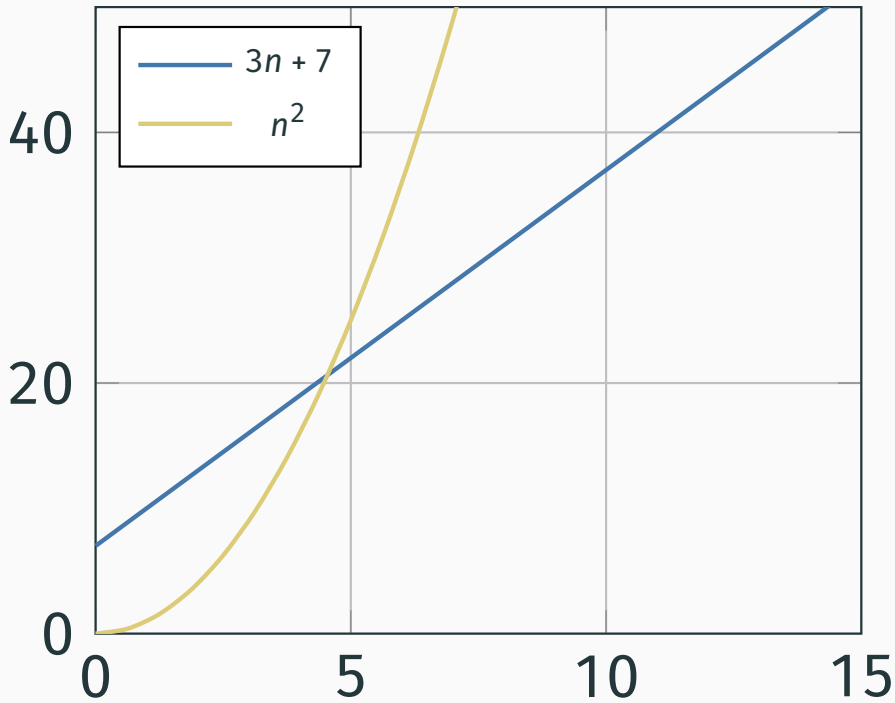
Řešení

1. Hledáme konstanty c a n_0 takové, aby platilo

$$3n + 7 \leq cn^2$$

pro všechna $n \geq n_0$.

2. Zvolíme-li $c = 1$, pak $n_0 = 5$.



O-notace – příklad 3

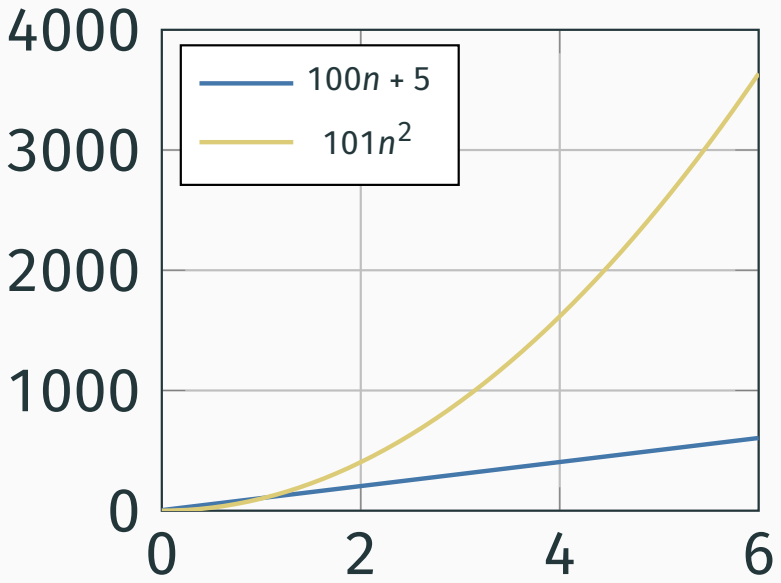
Dokažte, že $100n + 5 \in O(n^2)$.

Řešení

1. Platí, že $100n + 5 \leq 100n + n$ pro všechna $n \geq 5$.
2. Dále platí, že $101n \leq 101n^2$.
3. Odtud

$$100n + 5 \leq 101n \leq 101n^2$$

a tedy $c = 101$ a $n_0 = 5$.



O-notace – příklad 3 (pokrač.)

Ve výběru konstant c a n_0 máme ale mnoho možností – lze postupovat i takto:

$$100n + 5 \leq 100n + 5n = 105n$$

pro všechna $n \geq 1$. Z toho plyne, že

$$105n \leq 105n^2$$

a tudíž $c = 105$ a $n_0 = 1$.

Ω-notace

Definice

Mějme funkce $t(n)$ a $g(n)$, kde $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. Říkáme, že funkce $t(n)$ patří do $\Omega(g(n))$, jestliže existuje kladná nenulová reálná konstanta c a přirozené číslo $n_0 > 1$ takové, že

$$t(n) \geq cg(n)$$

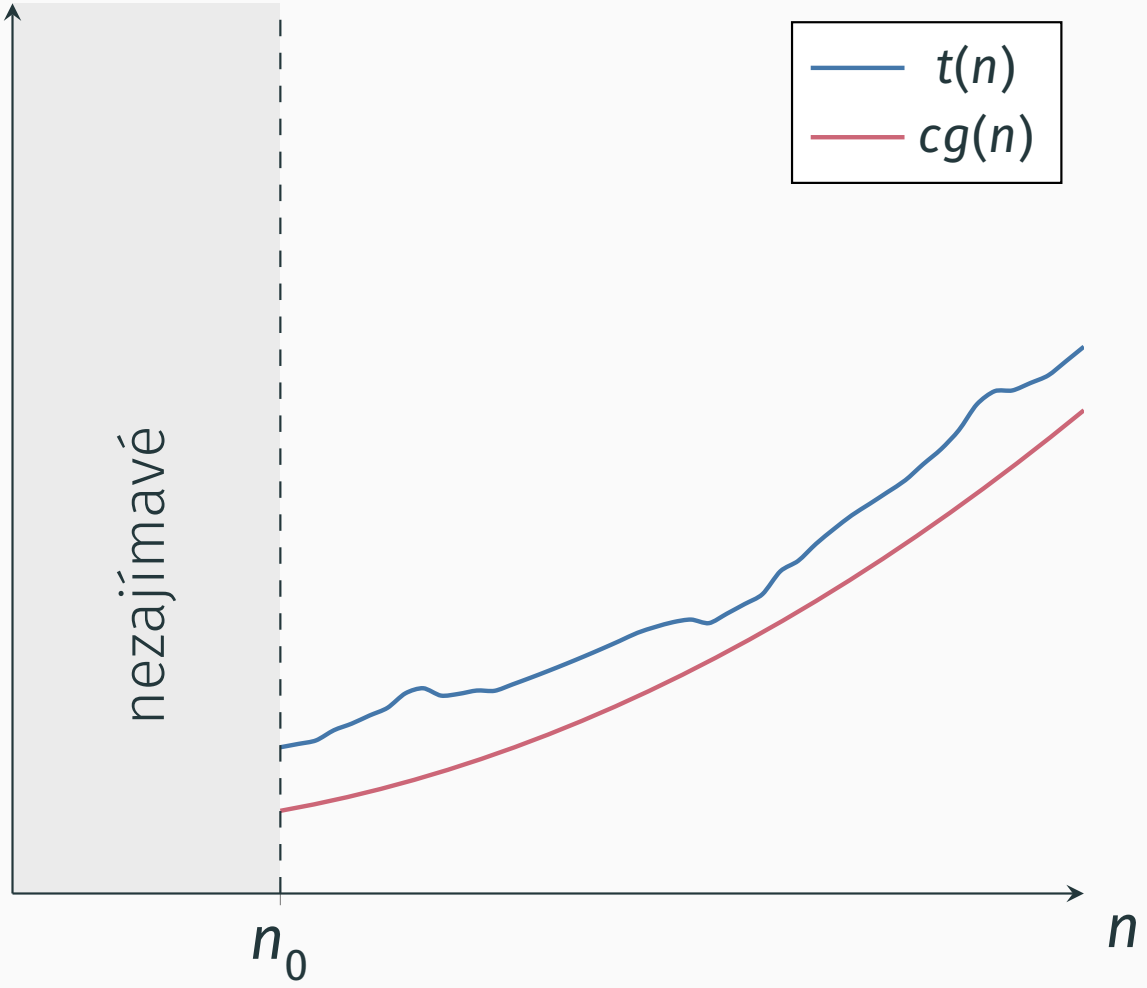
pro všechna $n \geq n_0$.

Příklad

Dokažte, že platí $n^3 \in \Omega(n^2)$. Zřejmě platí, že $n^3 \geq n^2$ pro všechna $n \geq 0$. Tudíž můžeme volit $c = 1$ a $n_0 = 1$.

Ω-notace graficky

$$t(n) \in \Omega(g(n))$$



Θ-notace

Definice

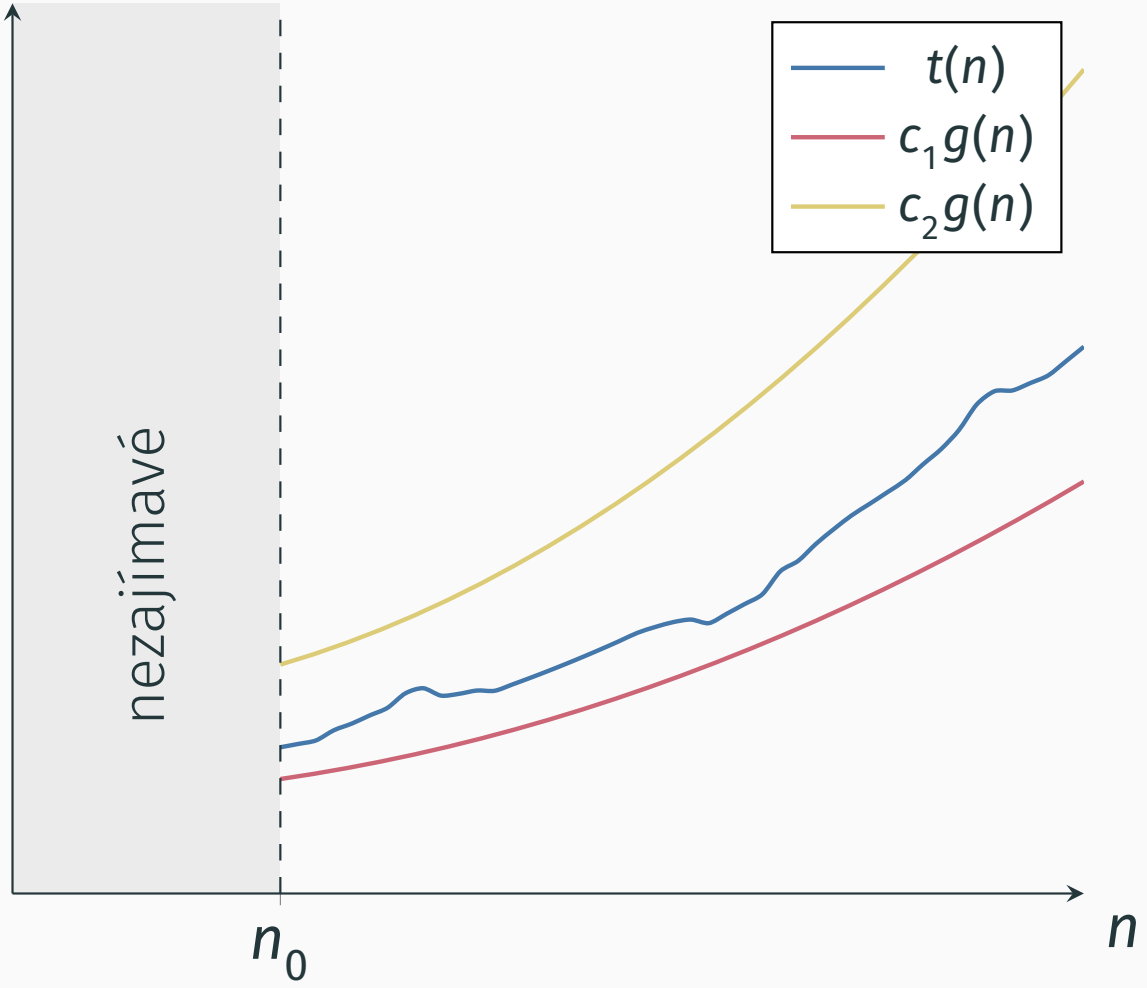
Mějme funkce $t(n)$ a $g(n)$, kde $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. Říkáme, že funkce $t(n)$ patří do $\Theta(g(n))$, jestliže existují kladné nenulové reálné konstanty c_1, c_2 a přirozené číslo $n_0 > 1$ takové, že

$$c_1 g(n) \leq t(n) \leq c_2 g(n)$$

pro všechna $n \geq n_0$.

Θ-notace graficky

$$t(n) \in \Theta(g(n))$$



Θ-notace – příklad

Zadání

Dokažte, že $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Řešení

1. Nejprve dokážeme pravou nerovnost $t(n) \leq c_2 g(n)$ (omezení shora)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$$

pro všechna $n \geq 0$.

Θ-notace – příklad (pokrač.)

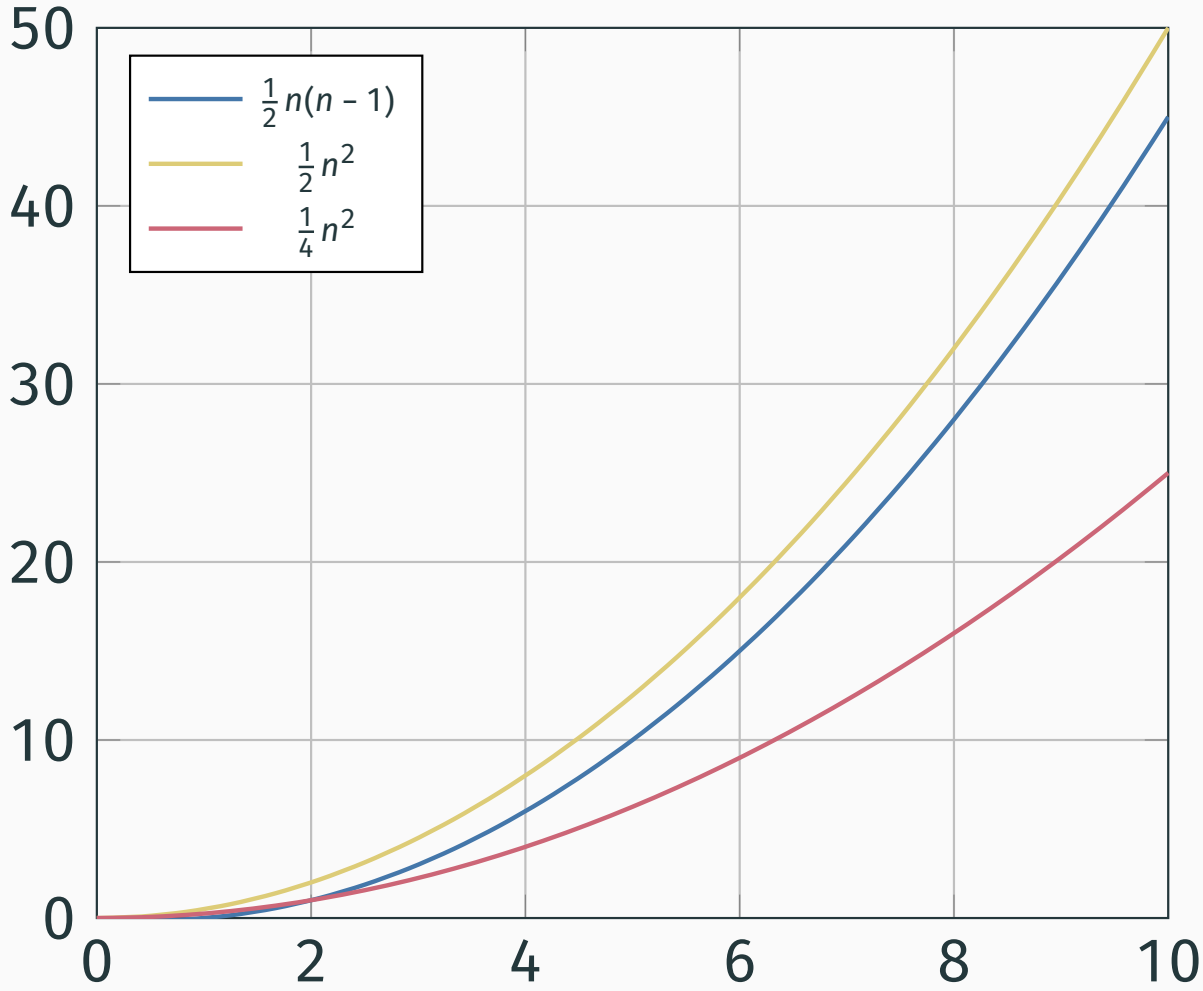
2. Levou nerovnost $c_1 g(n) \leq t(n)$ (omezení zdola) dokážeme takto:

$$\begin{aligned} t(n) = \frac{1}{2}n(n-1) &= \frac{1}{2}n^2 - \frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{4}n^2 \\ &\geq \frac{1}{4}n^2 \end{aligned}$$

   Souhrnně tedy $\frac{1}{4}n^2 \leq \frac{1}{2}n(n-1)$ pro všechna $n \geq 2$.

3. Z předchozích nerovností plyne, že $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$ a $n_0 = 2$.

Θ-notace – příklad (pokrač.)



Vlastnosti asymptotické notace

Základní vlastnosti:

1. $f(n) \in O(f(n))$
2. $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
3. $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \implies f(n) \in O(h(n))$
   tranzitivita, srovnej s $a \leq b \wedge b \leq c \implies a \leq c$

Vlastnosti asymptotické notace (pokrač.)

Problém

- Algoritmus A se skládá z částí A_1 a A_2.
- Části algoritmu se vykonávají po sobě, tj. po dokončení A_1 se začne vykonávat A_2.
- Složitost části A_1 je $t_1(n) \in O(g_1(n))$, složitost části A_2 je $t_2(n) \in O(g_2(n))$.
- Otázka zní – jaká je celková složitost algoritmu A?

Vlastnosti asymptotické notace (pokrač.)

Věta

Pokud $t_1(n) \in O(g_1(n))$ *a současně* $t_2(n) \in O(g_2(n))$*, potom*

$$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n))).$$

Poznámka

Shodné tvrzení můžeme vyslovit i pro Ω a Θ notaci.

Vlastnosti asymptotické notace (pokrač.)

Důkaz.

Protože $t_1(n) \in O(g_1(n))$, tak existuje kladná nenulová konstanta c_1 a nezáporná konstanta n_1 taková, že

$$t_1(n) \leq c_1 g_1(n) \qquad \forall n \geq n_1.$$

Obdobně

$$t_2(n) \leq c_2 g_2(n) \qquad \forall n \geq n_2.$$

Vlastnosti asymptotické notace (pokrač.)

Důkaz.

Označme $c_3 = \max(c_1, c_2)$ a $n_0 \geq \max(n_1, n_2)$. Potom platí

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq 2c_3 \max(g_1(n), g_2(n)). \end{aligned}$$

Tudíž $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$, protože existují konstanty $c = 2c_3 = 2\max(c_1, c_2)$ a $n_0 = \max(n_1, n_2)$. □

Závěr: Celkovou složitost algoritmu určuje část algoritmu s nejvyšší složitostí.

Vlastnosti asymptotické notace (pokrač.)

Příklad

Zadání: Test, zda se v poli vyskytují dvě shodné hodnoty.
Řešení:

1. Setřídění pole nevyžaduje ne více než $\frac{1}{2}n(n-1)$ porovnání, tj. složitost třídy $O(n^2)$.
2. Porovnání všech dvojic sousedních prvků bude vyžadovat $n-1$ porovnání, tj. složitost třídy $O(n)$.

Celková složitost algoritmu je tedy $O(\max(n^2, n)) = O(n^2)$.

Využití limit k výpočtům

Rychlost růstu funkcí lze snadněji počítat pomocí limit:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ roste pomaleji než } g(n) \\ c & t(n) \text{ roste stejně rychle jako } g(n) \\ \infty & t(n) \text{ roste rychleji než } g(n) \end{cases}$$

Je zřejmé, že:

$t(n) \in O(g(n)) \quad \Leftrightarrow \quad t(n)$ roste pomaleji nebo stejně rychle než $g(n)$

$t(n) \in \Omega(g(n)) \quad \Leftrightarrow \quad t(n)$ roste stejně rychle nebo rychleji než $g(n)$

$t(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad t(n)$ roste stejně rychle jako $g(n)$

Využití limit k výpočtům (pokrač.)

Některé užitečné vzorce

L'Hospitalovo pravidlo

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirlingův vzorec

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Využití limit k výpočtům – příklad I

Srovnejte rychlost růstu funkcí $\frac{1}{2}n(n-1)$ a n^2.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} \\
&= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) \\
&= \frac{1}{2}\left(\lim_{n\to\infty}1-\lim_{n\to\infty}\frac{1}{n}\right) \\
&= \frac{1}{2}(1-0) = \frac{1}{2} > 0
\end{aligned}$$

Funkce $\frac{1}{2}n(n-1)$ a n^2 rostou stejně rychle, tedy

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

.

Využití limit k výpočtům – příklad II

Srovnejte rychlost růstu funkcí $\log_2 n$ a $\sqrt{n}$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} &= \lim_{n\to\infty} \frac{\left(\log_2 n\right)'}{\left(\sqrt{n}\right)'} \\
&= \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = (\log_2 e) \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\
&= \log_2 e \lim_{n\to\infty} \frac{2\sqrt{n}}{n} \\
&= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0
\end{aligned}$$

Funkce $\log_2 n$ tedy **roste pomaleji** než $\sqrt{n}$.

Využití limit k výpočtům – příklad III

Srovnejte rychlost růstu funkcí $n!$ a 2^n.

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{2^n} &= \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\frac{n^n}{2^n e^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\left(\frac{n}{2e}\right)^n = \infty\end{aligned}$$

Poznámky

- Funkce $n!$ tedy roste rychleji než 2^n.
- Definice Θ-notace nevylučuje, že $n! \in \Omega(2^n)$, ale výpočet podle limity jasně říká, že $n!$ roste rychleji než 2^n

Základní třídy složitosti

Přestože teoreticky existuje nekonečně mnoho tříd složitosti, složitost většiny algoritmů padne do několika málo tříd.

Třída	Jméno	Poznámka
1	konstantní	složitost nezávisí na velikosti vstupu; jen velmi málo algoritmů
$\log n$	logaritmická	typicky algoritmy redukující velikost vstupu konstantním faktorem; vyhledávání půlením intervalu
n	lineární	algoritmy zpracovávající seznam o n prvcích; např. sekvenční vyhledávání
$n \log n$	lineárně-logaritmická	„rozděl a panuj“ algoritmy; průměrné složitosti QuickSortu, MergeSortu

Základní třídy složitosti (pokrač.)

Třída	Jméno	Poznámka
n^2	kvadratická	obecně algoritmy se dvěma vnořenými cykly; elementární metody třídění, sčítání matic typu $n \times n$
n^3	kubická	obecně algoritmy se třemi vnořenými cykly; násobení matic typu $n \times n$
2^n	exponenciální	typicky generování všech podmnožin n prvkové množiny
$n!$	faktoriál	typicky generování všech permutací n prvkové množiny

Vliv multiplikativní konstanty

- Třída složitosti je dána až na multiplikativní konstantu, která obvykle není přesně specifikována.
- Mohl by tedy algoritmus s vyšší třídy složitosti běžet, pro nějaké rozumné n, rychleji než algoritmus z lepší třídy? Například:

Algoritmus	Doba běhu
A	n^3
B	$10^6 n^2$

A bude lepší než B pro $n < 10^6$.

- Multiplikativní konstanty obvykle nabývají podobných, relativně malých, hodnot.
- **Lze očekávat, že algoritmy s nižší složitostí budou lepší než ty vyšší složitostí už pro středně velké vstupy.**

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.2, strany 52 – 61
- Kniha [3], kapitoly 3.1 a 3.2, strany 49 – 63

Analýza složitosti algoritmů

Analýza nerekurzivních algoritmů

Nalezení největšího prvku v poli n čísel

Vstup : Pole $A[0 \ldots n-1]$ celých čísel
Výstup: Největší prvek pole A

```
1 max ← A[0];
2 for i ← 1 to n - 1 do
3 |   if A[i] > max then
4 |   |   max ← A[i];
5 |   end
6 end
7 return max;
```

Nalezení největšího prvku v poli n čísel (pokrač.)

Pracovní postup

1. Velikost vstupu – velikost pole n
2. Základní operace:
   - nejčastěji vykonávané operace jsou uvnitř cyklu – porovnání $A[i] > max$ a přiřazení $max \leftarrow A[i]$
   - základní operací bude **porovnání**, protože se
     - provede v každém průchodu cyklem,
     - je to klíčová operace pro algoritmus, „Kolik dvojic prvků musím porovnat, abych našel maximum?“
3. Počet porovnání je stejný pro všechny vstupy velikosti n, není nutné rozlišovat mezi nejlepším, průměrným a nejhorším případem

Nalezení největšího prvku v poli n čísel (pokrač.)

4. Počet základních operací, porovnání, $C(n)$ bude roven

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

5. **Závěr**: Nalezení největšího prvku v poli n čísel je **lineární algoritmus**.

Nalezení největšího prvku v poli n čísel, všechny operace

Počet operací	Popis
1	přiřazení $max \leftarrow A[0]$
1	přiřazení $i \leftarrow 1$
$n - 1$	porovnání $i \leq n - 1$
$n - 1$	zvýšení i o 1
$n - 1$	porovnání $A[i] > max$
$n - 1$	přiřazení $max \leftarrow A[i]$
1	vrácení výsledku **return** max
$4(n-1)+3 = 4n - 1 \in \Theta(n)$	

Závěr: Nalezení největšího prvku v poli n čísel je **lineární algoritmus**.

Obecný postup určení časové složitosti nerek. algoritmů

1. Volba parametru, či parametrů, reprezentujícího velikost vstupu n.
2. Nalezení základních operací algoritmu (jsou to ty v nejvíce vnořeném cyklu!).
3. Závisí počet základních operací jen na velikosti vstupu? Pokud závisí i na něčem dalším, musíme zkoumat nejhorší, nejlepší a průměrný případ zvlášť.
4. Sestavení vztahu, resp. vztahů, („vzorečků“) vyjadřujících počet, resp. počty, provedení základních operací.
5. Zjednodušení sestavených vztahů a, nebo aspoň, stanovení řádového růstu.

Užitečné součtové vzorce

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i \tag{3}$$

$$\sum c a_i = c \sum a_i \tag{4}$$

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i \tag{5}$$

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1 \tag{6}$$

Konkrétně

$$\sum_{i=1}^{n} 1 = n \in \Theta(n) \tag{7}$$

Užitečné součtové vzorce (pokrač.)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1) \approx \frac{1}{2}n^2 \in \Theta(n^2) \tag{8}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1) \approx \frac{1}{3}n^3 \in \Theta(n^3) \tag{9}$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}, \text{ pro } a \neq 1 \tag{10}$$

Konkrétně

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n) \tag{11}$$

Unikátnost prvků v poli

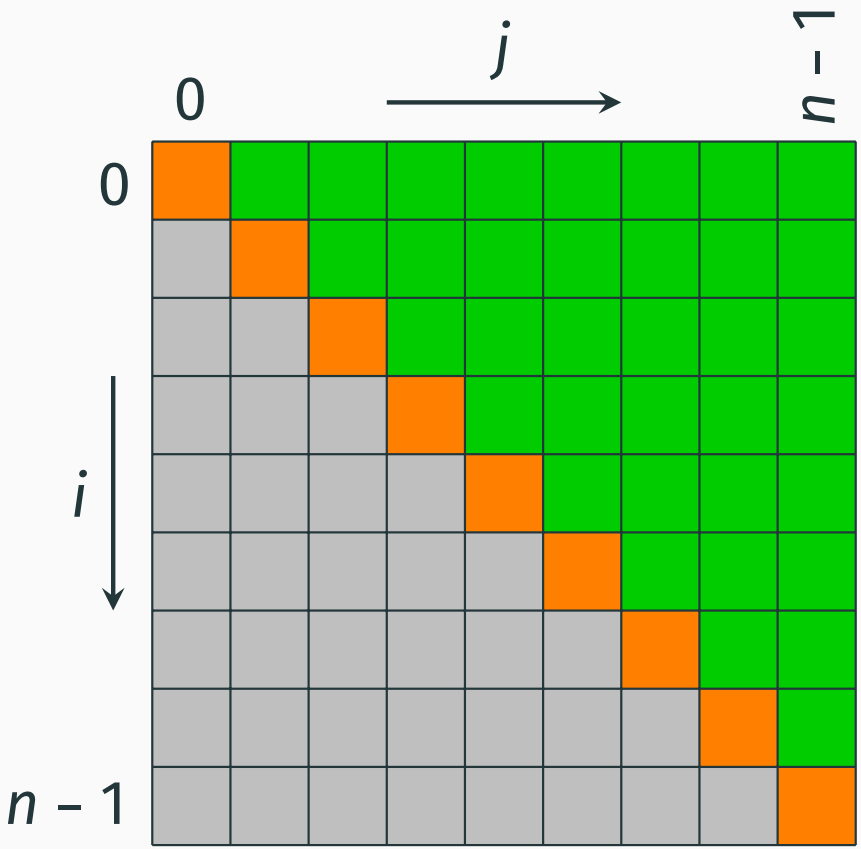
Je dáno pole o n prvcích. Naším úkolem provést analýzu algoritmu, který zjistí, zda všechny prvky v poli jsou navzájem různé, čili unikátní.

Vstup : Pole $A[0 \ldots n-1]$
Výstup: Vrací true, pokud jsou všechny prvky unikátní, jinak vrací false

```
1 for i ← 0 to n − 2 do
2 |   for j ← i + 1 to n − 1 do
3 |   |   if A[i] = A[j] then
4 |   |   |   return false;
5 |   |   end
6 |   end
7 end
8 return true;
```

Unikátnost prvků v poli (pokrač.)

Vizualizace algoritmu



Legenda

dvojice, které **je nutné** otestovat

prvek sám se sebou není nutné testovat

dvojice testované v předchozích průchodech cyklem

Unikátnost prvků v poli (pokrač.)

Pracovní postup

1. Velikost vstupu – velikost pole n
2. Základní operace – nejvíce vnořený cyklus obsahuje jedinou operaci, porovnání $A[i] = A[j]$
3. Závislost pouze na n? Ne, počet zákl. operací závisí i na tom, zda se v poli objeví shodný prvek. Tudíž provádíme analýzu **nejhoršího**, nejlepšího a průměrného případu.
4. Sestavení vztahů. Pro nejhorší případ je z vnitřního cyklu je patrné, že nesmí dojít k předčasnému ukončení cyklu, a to buď:
   - 4.1 protože všechny prvky jsou unikátní nebo
   - 4.2 je shodná až poslední dvojice.

Unikátnost prvků v poli (pokrač.)

Tudíž provedeme:

- jedno porovnání pro každý průchod vnitřním cyklem, tj. $j = i + 1, \ldots, n - 1$
- vnější cyklus, v každém svém průchodu, zopakuje celý vnitřní cyklus , tj. $i = 0, \ldots, n - 2$

Unikátnost prvků v poli (pokrač.)

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \qquad \text{podle (6)}$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \qquad \text{podle (3)}$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \qquad \text{podle (4) a (8)}$$

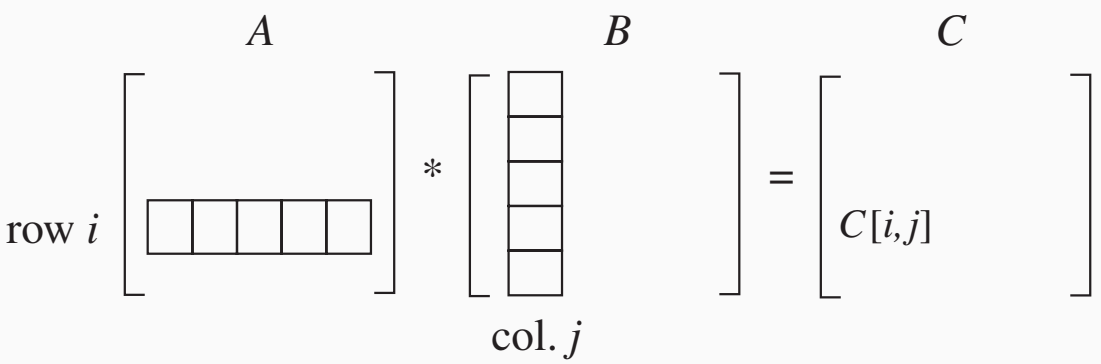
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} \qquad \text{podle (6)}$$

$$= \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Násobení čtvercových matic

Naším úkolem je provést analýzu algoritmu pro výpočet součinu $C = AB$ dvou čtvercových matic A a B řádu n.

Z definice jsou prvky matice rovny skalárním součinům řádků matice A se sloupci matice B.



$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

pro všechna
$0 \leq i, j \leq n - 1$

$$C[i,j] = A[i,0] \times B[0,j] + \cdots + A[i,k] \times B[k,j] + \cdots + A[i,n-1] \times B[n-1,j]$$

Násobení čtvercových matic (pokrač.)

Vstup : Dvě čtvercové matice A a B řádu n
Výstup: Čtvercová matice C řádu n, kde $C = AB$

```
1 for i ← 0 to n – 1 do
2 |   for j ← 0 to n – 1 do
3 |   |   C[i, j] ← 0;
4 |   |   for k ← 0 to n – 1 do
5 |   |   |   C[i, j] ← C[i, j] + A[i, k] × B[k, j];
6 |   |   end
7 |   end
8 end
9 return C;
```

Násobení čtvercových matic (pokrač.)

Pracovní postup

1. Velikost vstupu – řád matice n
2. Základní operace:
   - nejvíce vnořený cyklus obsahuje dvě operace – sčítání a násobení,
   - v každém průchodu se obě provedou přesně jedenkrát,
   - historická tradice velí počítat násobení (bývalo mnohokrát pomalejší než sčítání),
   - zavedeme $M(n)$ jako celkový počet násobení.
3. Počet operací závisí pouze na n – nejhorší, nejlepší a průměrný případ splývají

Násobení čtvercových matic (pokrač.)

4. Sestavení vztahů – počet násobení v nejvnitřnějším cyklu

$$\sum_{k=0}^{n-1} 1$$

Celkový počet násobení je roven

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Násobení čtvercových matic (pokrač.)

Pomocí vztahů (4) a (6) postupně dostáváme

$$
\begin{aligned}
M(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (n - 1 - 0 + 1) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = n \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \\
&= n \sum_{i=0}^{n-1} (n - 1 - 0 + 1) = n \sum_{i=0}^{n-1} n = n^2 \sum_{i=0}^{n-1} 1 \\
&= n^2 (n - 1 - 0 + 1) \\
&= n^3
\end{aligned}
$$

Násobení čtvercových matic (pokrač.)

Neformální postup:

1. algoritmus musí vypočítat $n \times n$ prvků matice C
2. každý prvek matice C je vypočten jako skalární součin i-tého řádku matice A a j-tého sloupce matice B
3. řádky i sloupce mají n prvků, které musíme vynásobit
4. celkem tedy $n^2 \times n = n^3$ násobení

Násobení čtvercových matic (pokrač.)

Doba běhu algoritmu na konkrétním počítači

$$T(n) \approx c_m M(n) = c_m n^3$$

započteme-li i sčítání

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

kde c_m resp. c_a je doba trvání násobení resp. sčítání, $A(n)$ je počet operací sčítání, platí $A(n) = M(n)$.

Shrnutí

Doba běhu algoritmu se může, v závislosti na konkrétním počítači, měnit, ale **řádová složitost algoritmu (n^3) zůstává**.

Počet bitů v binárním zápisu čísla

Naším úkolem je analyzovat algoritmus, který pro dané přirozené číslo n vypočte počet bitů nutných pro zápis čísla n v binární soustavě.

Vstup : Přirozené číslo n

Výstup: Počet bitů v binárním zápisu čísla n

```
1 count ← 1;
2 while n > 1 do
3 |   count ← count + 1;
4 |   n ← ⌊n/2⌋;
5 end
6 return count;
```

- Velikost vstupu – jedno číslo?

Počet bitů v binárním zápisu čísla (pokrač.)

- Základní operace – sčítání, dělení, porovnání s 1?
- Nejdůležitější je, v tomto případě, určit počet průchodů cyklem. Počet porovnání je o jedna větší než počet průchodů cyklem.
- Hodnota čísla n se v každém průchodu cyklem zmenšuje na polovinu, což vede ke vztahu

$$\lfloor \log_2 n \rfloor + 1$$

  a což odpovídá vztahu (2).
- K odvození budeme potřebovat umět řešit rekurzivní rovnice...

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.3, strany 61 – 70

Analýza složitosti algoritmů

Analýza rekurzivních algoritmů

Výpočet faktoriálu

Naším úkolem je analyzovat rekurzivní algoritmus, který pro dané přirozené číslo n vypočte jeho faktoriál $n!$.

$$n! = \begin{cases} 1 & \text{pro } n = 0 \\ n(n-1)! & \text{jinak} \end{cases}$$

```
1 Function F(n)
  |  Vstup: Přirozené číslo n
  |  Výsledek: Hodnota n!
2 |  if n = 0 then
3 |  |  return 1;
4 |  end
5 |  return n × F(n − 1);
6 end
```

Výpočet faktoriálu (pokrač.)

- Velikost vstupu – jedno číslo. Budeme brát v úvahu počet bitů? Ne, bylo by to komplikované.
- Základní operace – násobení. Alternativně lze uvažovat počet porovnání $n = 0$, které odpovídá počtu volání funkce F.
- Bude nás zajímat počet násobení $M(n)$ v závislosti na čísle n

Výpočet faktoriálu (pokrač.)

- Pro $n > 0$ se funkce $F(n)$ počítá jako

$$F(n) = F(n-1) \times n$$

a odtud

$$M(n) = M(n-1) + 1, \tag{12}$$

kde
$M(n-1)$ výpočet funkce $F(n-1)$ a
1 vynásobení výsledku $F(n-1)$ číslem n.

- Hodnota $M(n)$ není definována explicitně tj. jako funkce n například n^3, ale **implicitně** pomocí vztahu založeném na hodnotě totožné funkce pro jiné přirozené číslo, konkrétně $n-1$. Jde o tzv. **rekurentní (rekurzivní) vztah**.

Výpočet faktoriálu (pokrač.)

Poznámka

Explicitní	Implicitní
výslovný, přímý, jasný, zřetelný	zahrnutý, obsažený, ale nevyjádřený přímo
otevřeně, přímo vyjádřený, nenechávající nic zamlčeného, skrytého	nikoli zjevný, samo sebou se rozumějící
významově navzájem opačná slova	

Příklad

Explicitně: Lžete.
Implicitně: O pravdivosti vašeho tvrzení by se dalo s úspěchem pochybovat.

Výpočet faktoriálu (pokrač.)

- Cílem je najít najít explicitní vyjádření $M(n)$.
- Vztah $M(n) = M(n-1) + 1$ není jednoznačný, pro jednoznačné řešení je nutné definovat **počáteční podmínku**.
- Podmínka

  ```
  if n = 0 then
  |   return 1;
  end
  ```

  nám říká:

  1. nejmenší n pro které se algoritmus provede je $n = 0$ a
  2. v tomto případě algoritmus neprovede žádné násobení, tudíž $M(0) = 0$.

Výpočet faktoriálu (pokrač.)

- Celkově tedy pro výpočet $M(n)$ platí

$$\begin{aligned} M(n) &= M(n-1)+1 \text{ pro } n > 0 \\ M(0) &= 0 \end{aligned}$$

- Soustavu budeme řešit **metodou zpětné substituce**. Do

$$M(n) = M(n-1)+1$$

dosadíme za $M(n-1) = M(n-2)+1$

$$M(n) = [M(n-2)+1]+1 = M(n-2)+2$$

Výpočet faktoriálu (pokrač.)

a opět dosadíme za $M(n-2) = M(n-3) + 1$

$$M(n) = [M(n-3) + 1] + 2 = M(n-3) + 3.$$

Je zřejmé, že

$$M(n) = M(n-i) + i$$

Poznámka

Správnost této formule lze dokázat pomocí matematické indukce.

Výpočet faktoriálu (pokrač.)

Počáteční podmínka je definována pro $n = 0$, takže musíme dosadit za $i = n$ a dostáváme

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Shrnutí

1. Výsledek $M(n) = n$ byl víceméně očekávaný.
2. Iterativní algoritmus provede stejný počet násobení jak rekurzivní navíc bez režie spojené se zásobníkem pro volání funkcí.
3. Důležitý je ale popsaný přístup jak řešit rekurentní rovnice.

Obecný postup určení časové složitosti rekurzivních algoritmů

1. Volba parametru, či parametrů, reprezentujícího velikost vstupu n.
2. Nalezení základních operací algoritmu.
3. Závisí počet základních operací jen na velikosti vstupu? Pokud závisí i na něčem dalším, musíme zkoumat nejhorší, nejlepší a průměrný případ zvlášť.
4. Sestavení rekurentního vztahu a vhodné počáteční podmínky, vyjadřující počet provedení základních operací.
5. Zjednodušení sestavených vztahů a, nebo aspoň, stanovení řádového růstu.

Hanojské věže (Tower of Hanoi)

- Matematický hlavolam, autor Édouard Lucas, 1883.
- Hlavolam se skládá ze tří tyčí.
- Na začátku je na jedné tyči nasazeno několik kotoučů různých poloměrů, seřazených od největšího (vespod) po nejmenší (nahoře).

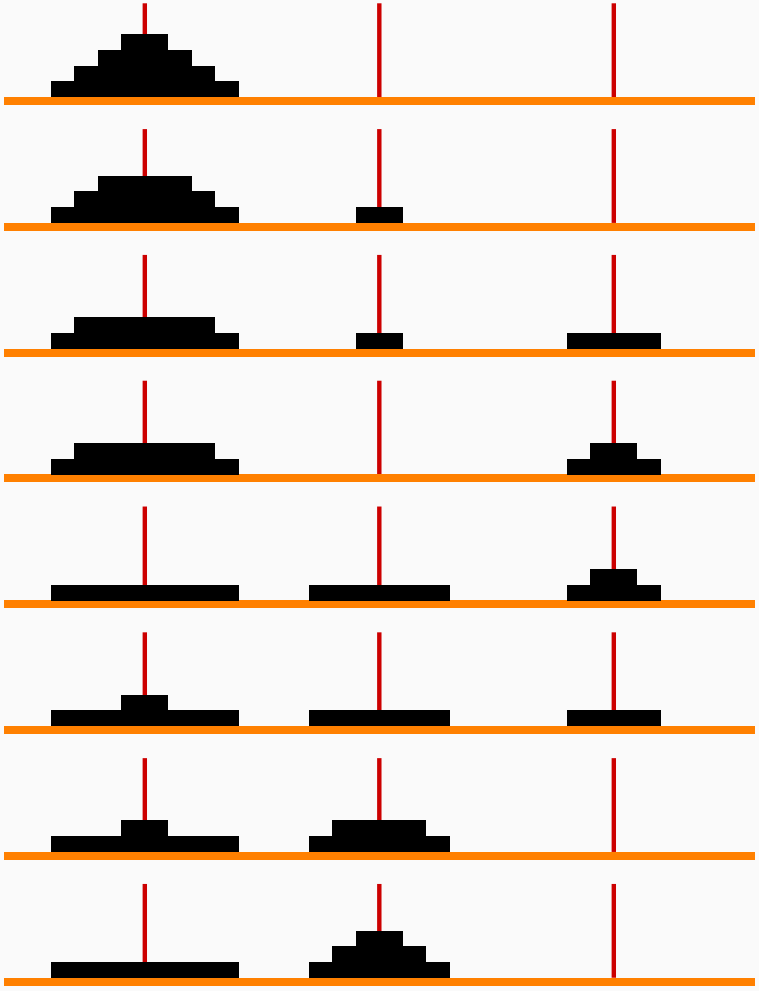
- Úkolem je přemístit všechny kotouče z první tyče na třetí tyč za pomoci druhé tyče.
- Pravidla hry:
  - V jednom tahu lze přemístit jen jeden kotouč.

Hanojské věže (Tower of Hanoi) (pokrač.)

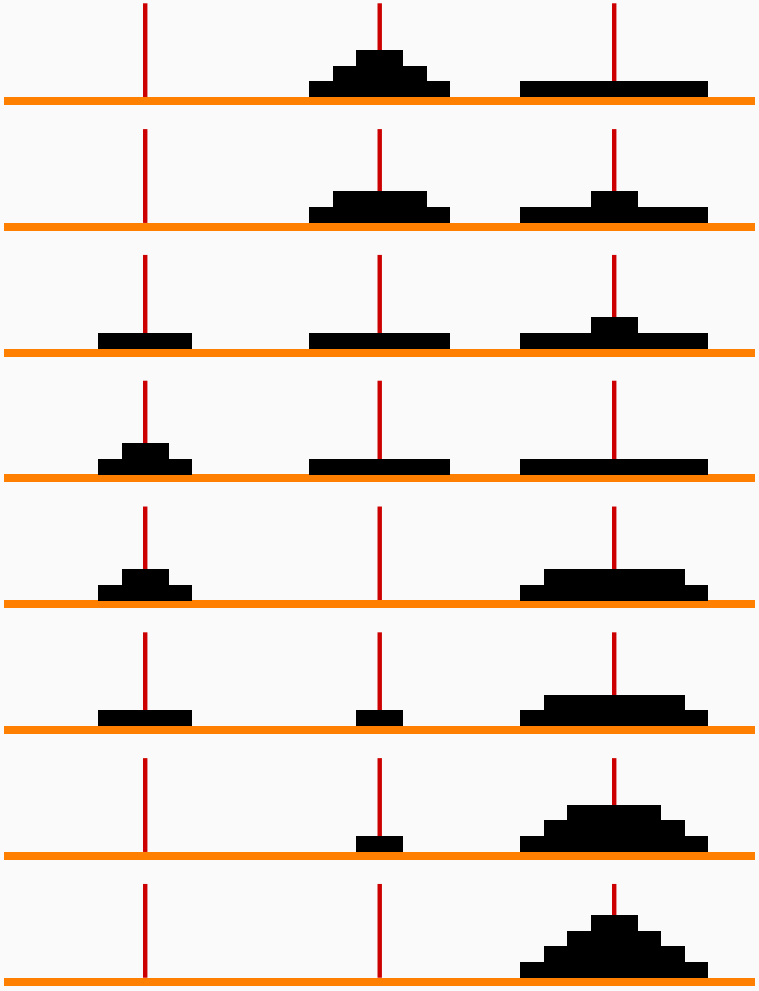
  - Jeden tah se sestává ze sejmutí vrchního kotouče z některé tyče a jeho navlečení na jinou tyč.
  - Je zakázáno položit větší kotouč na menší.
- Podle legendy stojí v Hanoji klášter, v němž jsou hanojské věže se 64 zlatými kotouči. Mniši každý den v poledne přemístí jeden kotouč. V okamžiku, kdy bude přemístěn poslední kotouč, nastane konec světa.
- Hlavně klid! Řešení tohoto hlavolamu pro 64 kotoučů vyžaduje $2^{64} - 1 = 18\,446744073709551615$ tahů. I kdyby přemístili každou sekundu jeden kotouč (a postupovali nejkratším možným způsobem), doba řešení je cca 600 miliard let.

Hanojské věže (Tower of Hanoi) (pokrač.)

Kroky řešení 1 – 8



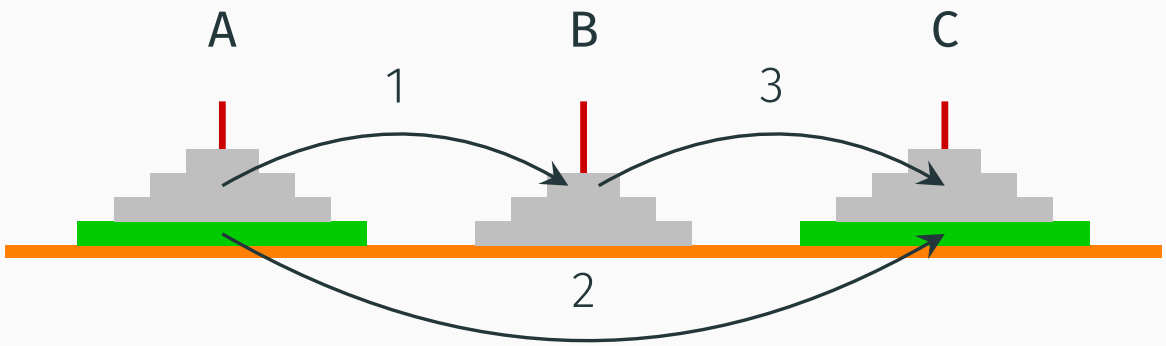
Kroky řešení 9 – 16



Hanojské věže (Tower of Hanoi) (pokrač.)

Řešení problému pro n disků

1. Přesun $n - 1$ disků z tyče A na tyče B (pomocí tyče C).
2. Přesun největšího disku z tyče A přímo na tyč C.
3. Přesun $n - 1$ disků z tyče B na tyč C (pomocí tyče A).



Pro $n = 1$ existuje triviální řešení...

Hanojské věže (Tower of Hanoi) (pokrač.)

Pracovní postup

1. Velikost vstupu – počet disků n.
2. Základní operace – přesun disku.
3. Závislost pouze na n? Ano. Tudíž **nemusíme** zkoumat nejhorší, nejlepší a průměrný případ zvlášť.
4. Sestavení vztahů

$$M(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2M(n-1) + 1 & \text{jinak} \end{cases} \tag{13}$$

Hanojské věže (Tower of Hanoi) (pokrač.)

5. Řešení metodou zpětné substituce. Do vztahu

$$M(n) = 2M(n-1) + 1$$

dosadíme $M(n-1) = 2M(n-2) + 1$

$$M(n) = 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1,$$

opět dosadíme $M(n-2) = 2M(n-3) + 1$

$$M(n) = 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1,$$

po dalším dosazení dostaneme

$$M(n) = 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$$

Hanojské věže (Tower of Hanoi) (pokrač.)

Po i-tém dosazení dostáváme

$$\begin{aligned} M(n) &= 2^i M(n-i) + \underbrace{2^{i-1} + 2^{i-2} + \cdots + 2 + 1}_{\text{sečteme podle (11)}} \\ &= 2^i M(n-i) + 2^i - 1. \end{aligned}$$

Počáteční podmínky platné pro $n = 1$ dosáhneme při $i = n - 1$

$$\begin{aligned} M(n) &= 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 \\ &= 2^{n-1} \cdot 1 + 2^{n-1} - 1 = 2 \cdot 2^{n-1} - 1 = 2^{n-1+1} - 1 \\ &= 2^n - 1 \end{aligned}$$

Hanojské věže (Tower of Hanoi) (pokrač.)

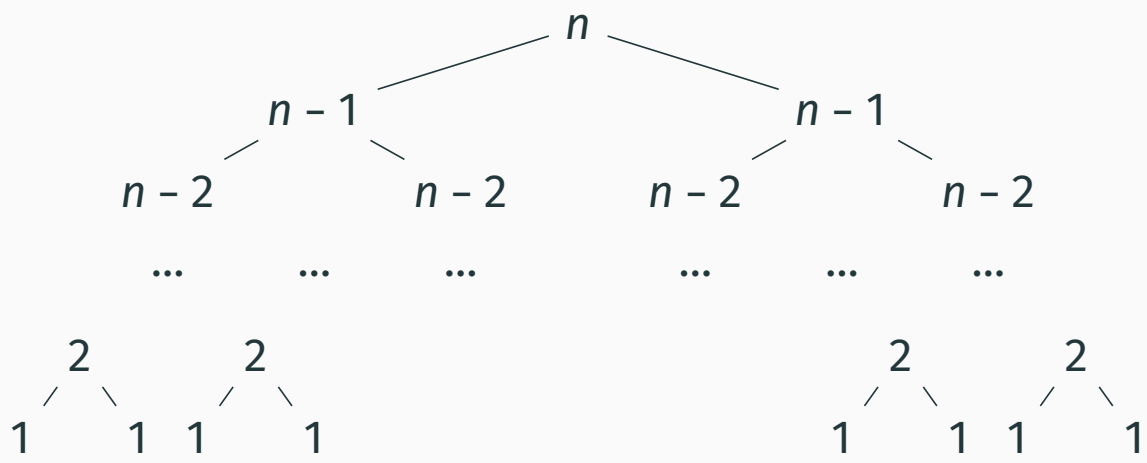
6. Závěr:
   - 6.1 Navržený rekurzivní algoritmus provede exponenciální počet základních operací vzhledem k velikosti vstupu.
   - 6.2 Algoritmus je použitelný jen pro malá n, což není způsobeno nevhodným návrhem. Je to způsobeno podstatou problému – lze dokázat, že toto je nejlepší možný algoritmus.

Opatrně s rekurzivními algoritmy

Obecně je nutné k rekurzivním algoritmům přistupovat velice opatrně, protože jejich stručnost může maskovat neefektivitu.

Hanojské věže (Tower of Hanoi) (pokrač.)

Vizualizace rekurzivního volání



Počet uzlů odpovídá počtu volání rekurzivní funkce

$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1,$$

kde l je číslo úrovně ve stromu.

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.4, strany 70 – 79

Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Strategie řešení problémů hrubou silou a úplným prohledáváním

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Strategie řešení problémů hrubou silou

Charakteristika

Strategie řešení problémů hrubou silou je založena na **přímočarém přístupu k řešení problému**, kdy algoritmus řešení vychází přímo ze zadání problému a pojmů obsažených v zadání.

Strategie řešení problémů hrubou silou – příklady

Výpočet mocniny

Máme vypočítat mocninu a^n pro nenulové a a n přirozené číslo. Z definice mocnění platí

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n \text{ krát}}$$

Řešení hrubou silou – vynásobíme číslo a mezi sebou $n - 1$ krát.

Další příklady řešení hrubou silou:

- výpočet NSD postupným dělením a
- násobení matic podle algoritmu z předchozí prezentace.

Strategie řešení problémů hrubou silou

1. obecná strategie – je obtížné najít problém, kde by „nezabrala“,
2. obecně sice nevede k efektivním algoritmům, ale pro některé problémy, např. násobení matic, pattern matching, jsou algoritmy založené na této strategii použitelné i pro větší vstupy,
3. přijatelná strategie v případě, kdy se nevyplatí zabývat se sofistikovanějším algoritmem – ad hoc řešení problému,
4. vždy použitelná strategie pro řešení problémů s malou velikostí vstupu a
5. význam i jako měřítko, kterým můžeme poměřovat efektivnější algoritmy řešící stejný problém.

Strategie řešení problémů hrubou silou a úplným prohledáváním

Třídící algoritmy

Třídící algoritmy

- Máme dáno pole n prvků pro které je definována relace uspořádání (čili vztah „menší než“), pro ukázku vezměme celé čísla.
- Úkolem je přeuspořádání prvků pole do neklesající posloupnosti – prvek pole na nižším indexu musí být menší nebo roven prvku na vyšším indexu.
- Otázka zní: existuje třídící algoritmus řešící problém hrubou silou, zcela přímočaře?

Třídění výběrem – SelectSort

Výchozí úvaha

Otázka: O kterém prvku z pole víme přesně kam patří?

Odpověď: O nejmenším! Patří na začátek pole, na nejnižší index! (Pozn. Totéž platí pro největší prvek.)

Princip algoritmu

1. Vybereme z n prvků pole nejmenší prvek a zaměníme jej s prvním prvkem pole.
2. Vybereme ze zbylých $n - 1$ prvků pole nejmenší prvek a zaměníme jej s druhým prvkem pole.
3. Obecně v i-tém kroku vybereme ze zbylých $n - i$ nejmenší prvek a zaměníme ho s i-tým prvkem.
4. Po $n - 1$ krocích je pole setříděno.

Třídění výběrem – ukázka

89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89
17	29	34	90	45	68	89
17	29	34	45	90	68	89
17	29	34	45	68	90	89
17	29	34	45	68	89	90

Třídění výběrem – pseudokód

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1 for i ← 0 to n - 2 do
2 |   min ← i;
3 |   for j ← i + 1 to n - 1 do
4 |   |   if A[j] < A[min] then
5 |   |   |   min ← j;
6 |   |   end
7 |   end
8 |   Swap (A[i], A[min]);
9 end
```

Funkce pro výměnu hodnot dvou proměnných

```
1 Procedure Swap(x, y)
  |   Vstup  : Parametry x a y shodného datového typu
  |   Výstup: Navzájem přehozené hodnoty x a y
2 |   aux ← x;
3 |   x ← y;
4 |   y ← aux;
5 end
```

Třídění výběrem – analýza

1. Velikost vstupu – počet prvků n.
2. Základní operace – porovnání prvků (někdy i počet výměn prvků).
3. Počet základních operací závisí pouze na n – nejhorší, nejlepší a průměrný případ splývají.
4. Sestavení vztahů

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 = \Theta(n^2)$$

Třídění výběrem – analýza (pokrač.)

Poznámky

- Podrobný výpočet sumy lze najít v příkladu „Unikátnost prvků“ v minulé lekci.
- Složitost algoritmu nijak nezávisí na míře nesetříděnosti vstupního pole. Algoritmus tedy **není přirozený**.
- Algoritmus je ale **stabilní** – jako minimum je vždy brán první prvek z několika shodných.
- Algoritmus je **in situ** – vystačíme si s konstantním rozsahem paměti navíc.

Třídění výběrem

Animace

K algoritmu třídění výběrem je k dispozici animace v samostatném souboru.

Bublinové třídění – BubbleSort

- Vezmu dvojici sousedních prvků A_i a A_{i+1}.
- Pokud jsou v nesprávném pořadí, tak je vyměním.
- Zvýším i o jedna a pokračuji další dvojicí prvků.
- Po každém průchodu polem A se dostane jeden prvek určitě na své místo.

$$A_0, \dots, A_j \leftrightarrow A_{j+1}, \dots A_{n-i-1} \mid \underbrace{A_{n-i} \leq \dots \leq A_{n-1}}_{\text{setříděno}}$$

  V dalším průchodu polem už procházím o jeden prvek méně.
- Proč bublinové třídění – největší prvek z nesetříděné části takto „probublá“ směrem ke konci pole.

Bublinové třídění – ukázka

89	↔	45		68		90		29		34		17
45		89	↔	68		90		29		34		17
45		68		89	↔	90		29		34		17
45		68		89		90	↔	29		34		17
45		68		89		29		90	↔	34		17
45		68		89		29		34		90	↔	17
45		68		89		29		34		17		90
45	↔	68	↔	89	↔	29		34		17		90
45		68		29		89	↔	34		17		90
45		68		29		34		89	↔	17		90
45		68		29		34		17		89		90

a tak dále

Bublinové třídění – pseudokód

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1 for i ← 0 to n – 2 do
2 |   for j ← 0 to n – i – 2 do
3 |   |   if A[j] > A[j + 1] then
4 |   |   |   Swap (A[j], A[j + 1]);
5 |   |   end
6 |   end
7 end
```

Bublinové třídění – analýza

1. Velikost vstupu – počet prvků n.
2. Základní operace – porovnání prvků.
3. Počet základních operací závisí pouze na n – nejhorší, nejlepší a průměrný případ splývají.
4. Sestavení vztahů

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} 1 = \sum_{i=0}^{n-2} [(n-i-2)-0+1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 = \Theta(n^2)$$

Bublinové třídění – analýza (pokrač.)

Počet základních operací $C(n)$ je shodný s tříděním výběrem.
Počet vzájemných výměn prvků $S(n)$ ale už na vstupu závisí a je v nejhorším případě roven

$$S_{worst}(n) = C(n) = \frac{1}{2}n(n-1) \in \Theta(n^2)$$

Bublinové třídění – další vlastnosti

- Algoritmus je **in situ** – vystačíme si s konstantním rozsahem paměti navíc.
- Algoritmus je **stabilní** – k výměně dojde jen pokud je $A[j] > A[j+1]$, jinak ne.
- Složitost této verze algoritmu nijak nezávisí na míře nesetříděnosti vstupního pole. Algoritmus tedy **není přirozený**.
- Modifikované verze (viz dále) už ale přirozené jsou.

Bublinové třídění – alternativní konstrukce algoritmu I

- Počet opakování vnějšího cyklu by měl záviset na provedení či neprovedení výměny prvků.
- Pokud se během vnitřního cyklu neprovedla žádná výměna prvků není nutné toto pole dále procházet – pole je evidentně setříděno.
- Zavedeme si logickou proměnnou *AnySwap*, kam si budeme ukládat informaci o případné výměně.

Bublinové třídění – alternativní konstrukce algoritmu I

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1 do
2 |   AnySwap ← false;
3 |   for i ← 0 to n – 2 do
4 |   |   if A[i] > A[i + 1] then
5 |   |   |   Swap (A[i], A[i + 1]);
6 |   |   |   AnySwap ← true;
7 |   |   end
8 |   end
9 while AnySwap;
```

Bublinové třídění – alternativní konstrukce algoritmu II

- Kombinovaná konstrukce algoritmu:
  - nedošlo k výměně prvků a
  - zkracování posloupnosti o jeden prvek zprava.
- Zavedeme si logickou proměnnou *AnySwap*, kam si budeme ukládat informaci o případné výměně.
- Proměnná *Right* bude označovat pravý okraj tříděné posloupnosti.

Bublinové třídění – alternativní konstrukce algoritmu II

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1  Right ← n – 2;
2  do
3  |   AnySwap ← false;
4  |   for i ← 0 to Right do
5  |   |   if A[i] > A[i + 1] then
6  |   |   |   Swap (A[i], A[i + 1]);
7  |   |   |   AnySwap ← true;
8  |   |   end
9  |   end
10 |   Right ← Right – 1;
11 while AnySwap;
```

Bublinové třídění – alternativní konstrukce algoritmu III

Proměnná *LastSwapIndex*

- bude označovat index poslední výměny prvků,
- jinak řečeno pravý okraj tříděné posloupnosti v následujícím průchodu polem.

Bublinové třídění – alternativní konstrukce algoritmu III

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1  Right ← n − 2;
2  do
3  |   LastSwapIndex ← 0;
4  |   for i ← 0 to Right do
5  |   |   if A[i] > A[i + 1] then
6  |   |   |   Swap (A[i], A[i + 1]);
7  |   |   |   LastSwapIndex ← i + 1;
8  |   |   end
9  |   end
10 |   Right ← LastSwapIndex;
11 while LastSwapIndex > 0;
```

Bublinové třídění

Animace

K algoritmu bublinového třídění je k dispozici animace v samostatném souboru. V animaci je znázorněn algoritmus podle snímku 339.

Třídění přetřásáním – ShakerSort

U BubbleSortu lze pozorovat dva efekty:

1. „velké“ prvky, **zajíci**, se velice rychle posunují ke konci pole, ale
2. „malé“ prvky, **želvy**, se posunují na začátek pole jen jako důsledek rychlého přesunu velkých prvků.

Modifikace BubbleSortu:

- pole budeme procházet střídavě z obou stran
- po každém průchodu pole se vymění role zajíců a želv, cimrmanovská „výměna mečů“ ze hry Blaník
- **ShakerSort** – pole je „přestřásáno, setřepáváno“ jako v barovém shakeru.

Třídění přetřásáním – ukázka

89	↔	45		68		90		29		34		17		
45		89	↔	68		90		29		34		17		
45		68		89	↔	90		29		34		17		
45		68		89		90	↔	29		34		17		
45		68		89		29		90	↔	34		17		
45		68		89		29		34		90	↔	17		
45		68		89		29		34		17		90		
45		68		89		29		34	↔	17		90		
45		68		89		29	↔	17		34		90		
45		68		89	↔	17		29		34		90		
45		68	↔	17		89		29		34		90		
45	↔	17		68		89		29		34		90		
17		45		68		89		29		34		90		

a tak dále

Třídění přetřásáním – pseudokód

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1 Left ← 0;
2 Right ← n - 1;
3 do
4 |   LeftToRight (A, Left, Right);
5 |   RightToLeft (A, Left, Right);
6 while Left < Right;
```

Třídění přetřásáním – pseudokód (pokrač.)

```
1  Procedure LeftToRight(A, Left, Right)
2      j ← 0;
3      for i ← Left to Right – 1 do
4          if A[i] > A[i + 1] then
5              Swap (A[i], A[i + 1]);
6              j ← i;
7          end
8      end
9      Right ← j;
10 end
```

Třídění přetřásáním – pseudokód (pokrač.)

```
1  Procedure RightToLeft(A, Left, Right)
2      j ← 0;
3      for i ← Right downto Left + 1 do
4          if A[i − 1] > A[i] then
5              Swap (A[i − 1], A[i]);
6              j ← i;
7          end
8      end
9      Left ← j;
10 end
```

Třídění přetřásáním

Animace

K algoritmu třídění přetřásáním je k dispozici animace v samostatném souboru.

Strategie řešení problémů hrubou silou a úplným prohledáváním

Sekvenční vyhledávání

Sekvenční vyhledávání

- Typická ukázka strategie řešení hrubou silou.
- Silná stránka algoritmu – jednoduchost (simplicity).
- Slabá stránka algoritmu – vysoká složitost.

Vstup : Pole $A[0 \dots n-1]$ a hledaný prvek x
Výstup: Index prvního výskytu prvku x v poli A, jinak -1

```
1 for i ← 0 to n − 1 do
2 |   if A[i] = x then
3 |   |   return i;
4 |   end
5 end
6 return -1;
```

Algoritmus bývá také označován jako **lineární vyhledávání**.

Lineární vyhledávání – složitost

	Počet porovnání
Nejhorší případ	n
Nejlepší případ	1
Průměrný případ	$\frac{1}{2}p(n+1) + n(1-p)$

kde p je pravděpodobnost úspěšného vyhledání

Lineární vyhledávání – využití zarážky (angl. sentinel)

Vstup : Pole $A[0 \dots n]$ a hledaný prvek x
Výstup: Index prvního výskytu prvku x v poli A, jinak -1

```
1 A[n] ← x;
2 i ← 0;
3 while A[i] ≠ x do
4 |   i ← i + 1;
5 end
6 if i < n then return i;
7 return -1;
```

Strategie řešení problémů hrubou silou a úplným prohledáváním

Vyhledávání podřetězce hrubou silou

Vyhledávání podřetězce hrubou silou

Zadání – najít vzorek p v textu t.

Řešení hrubou silou

1. Přiložíme vzorek na začátek textu.
2. Začneme porovnávat znaky ve vzorku a textu.
3. Pokud se shodují všechny znaky vzorku s textem – nalezeno.
4. Pokud nalezneme neshodu, posuneme vzorek o jednu pozici dopředu a pokračujeme bodem 2

$$\begin{array}{ccccccccc} t_0 & \cdots & t_i & \cdots & t_{i+j} & \cdots & t_{i+m-1} & \cdots & t_{n-1} \\ & & \updownarrow & & \updownarrow & & \updownarrow & & \\ & & p_0 & \cdots & p_j & \cdots & p_{m-1} & & \end{array}$$

Vyhledávání hrubou silou – pseudokód

Vstup : Vzorek p, text t a počáteční pozice s
Výstup: Pozice prvního výskytu p v textu t nebo -1

```
1 for i ← s to |t| - |p| do
2 |   j ← 0;
3 |   while j < |p| do
4 |   |   if p[j] ≠ t[i + j] then break;
5 |   |   j ← j + 1;
6 |   end
7 |   if j = |p| then return i;
8 end
9 return -1;
```

Vyhledávání hrubou silou – příklad

První pokus

Posun o 1 znak

Druhý pokus

Posun o 1 znak

Vyhledávání hrubou silou – příklad (pokrač.)

Třetí pokus

Posun o 1 znak

Čtvrtý pokus

Posun o 1 znak

Vyhledávání hrubou silou – příklad (pokrač.)

Pátý pokus

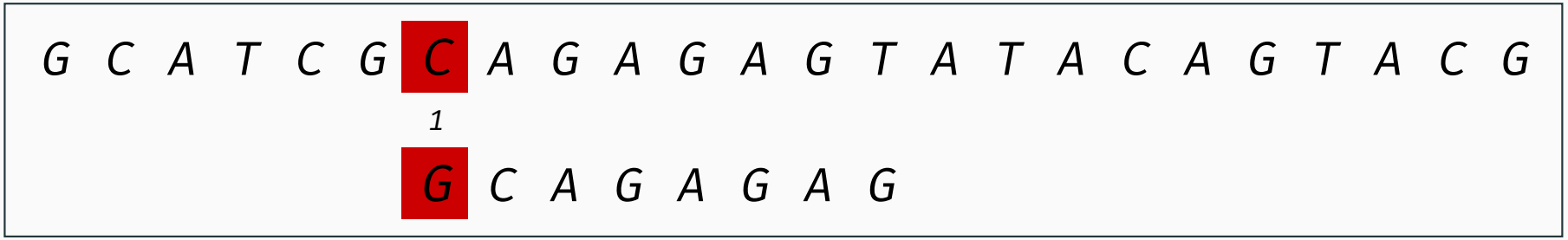
Posun o 1 znak

Šestý pokus

Posun o 1 znak

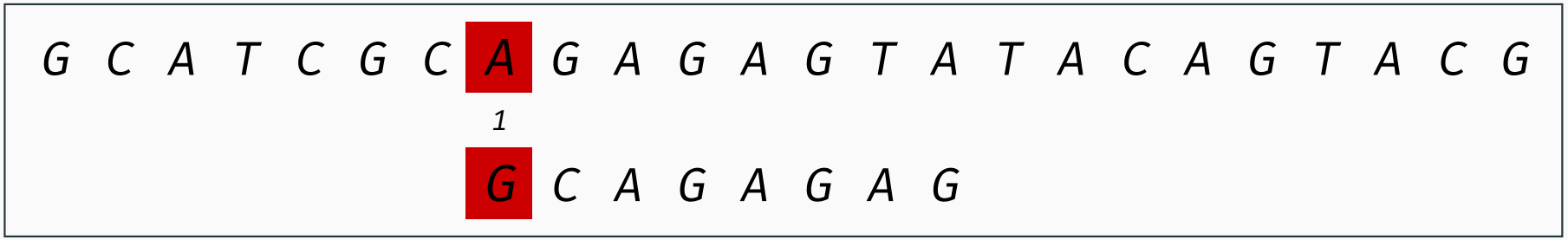
Vyhledávání hrubou silou – příklad (pokrač.)

Sedmý pokus



Posun o 1 znak

Osmý pokus



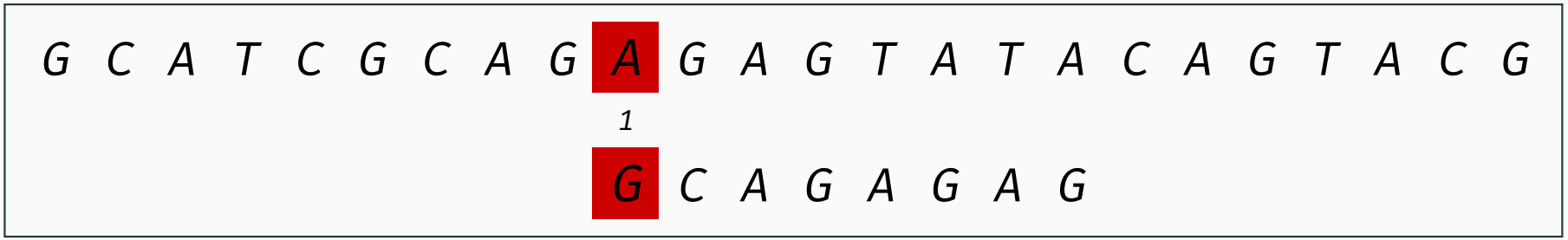
Posun o 1 znak

Vyhledávání hrubou silou – příklad (pokrač.)

Devátý pokus

Posun o 1 znak

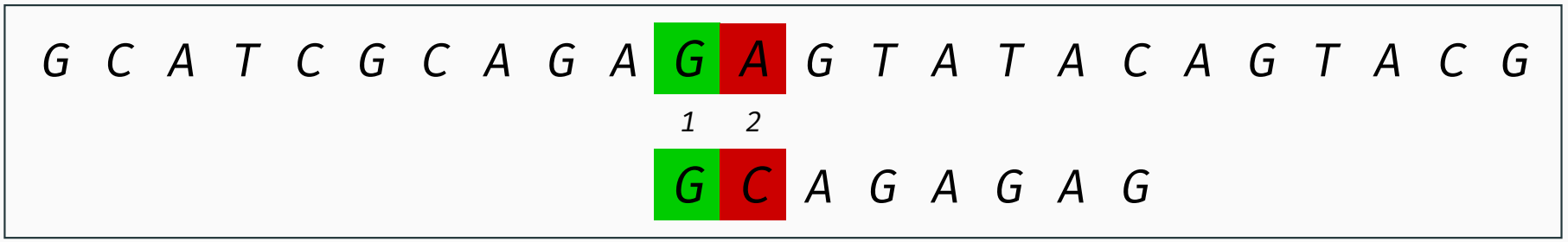
Desátý pokus



Posun o 1 znak

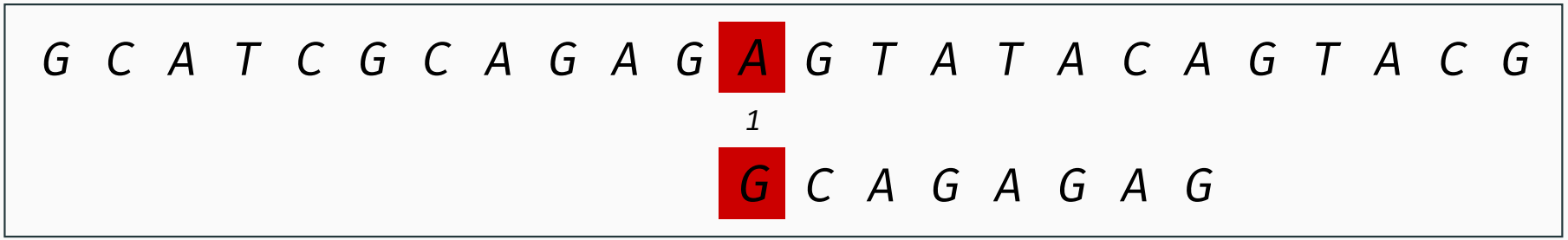
Vyhledávání hrubou silou – příklad (pokrač.)

Jedenáctý pokus



Posun o 1 znak

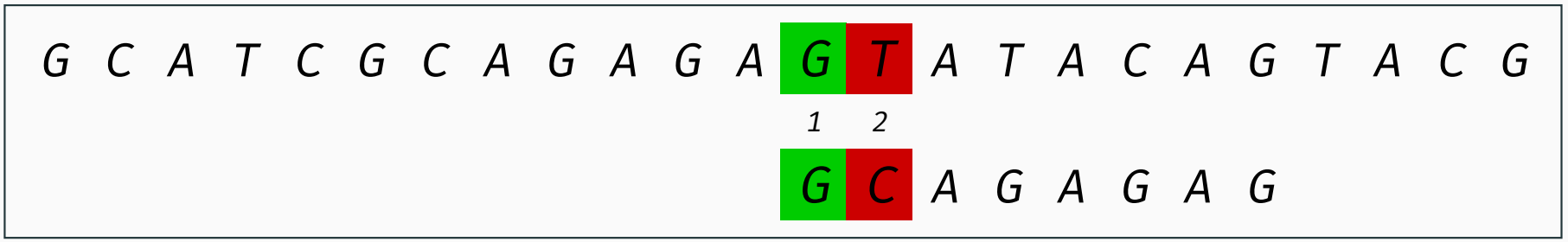
Dvanáctý pokus



Posun o 1 znak

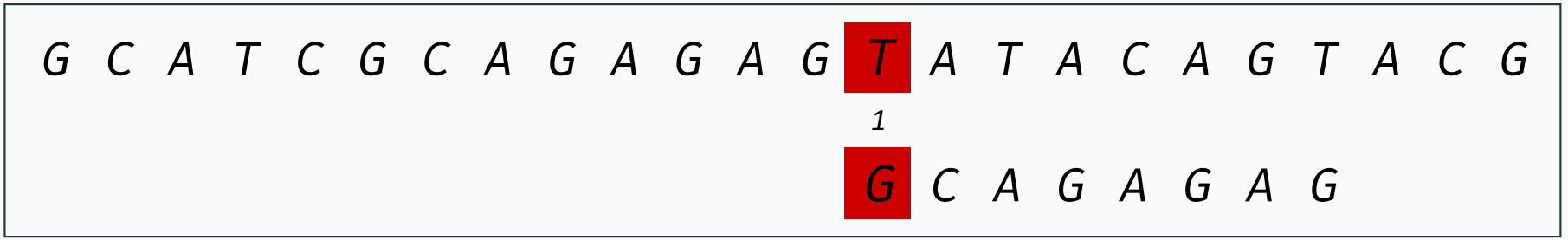
Vyhledávání hrubou silou – příklad (pokrač.)

Třináctý pokus



Posun o 1 znak

Čtrnáctý pokus



Posun o 1 znak

Vyhledávání hrubou silou – příklad (pokrač.)

Patnáctý pokus

Posun o 1 znak

Šesnáctý pokus

Posun o 1 znak

Vyhledávání hrubou silou – příklad (pokrač.)

Sedmnáctý pokus

Posun o 1 znak

Algoritmus provedl celkem 30 porovnání znaků.

Vyhledávání podřetězce hrubou silou – složitost algoritmu

- Velikost vstupu – délka textu n a délka vzorku m.
- Základní operace – porovnání znaku vzorku a textu.
- Závislost jen na velikosti vstupu – ne, záleží i na tom, kdy se najde první neshoda.
- Nejhorší případ – text $a^{n-1}b$, vzorek $a^{m-1}b$
  - v každém pokusu provedeme všech m porovnání vzorku s textem
  - současně provedeme všech $n - m + 1$ pokusů.
  - Celkem provedeme $m(n - m + 1)$ porovnání, algoritmus spadá do $O(mn)$.
- Nejlepší případ – vzorek je nalezen na začátku textu, složitost $O(m)$.
- Přirozené jazyky – posun nastane po „několika“ (k_L) porovnáních, nejhorší složitost $O(k_L n) = O(n)$.

Strategie řešení problémů hrubou silou a úplným prohledáváním

Problém nejbližší dvojice bodů

Problém nejbližší dvojice bodů

Zadání problému

Nalezněte dva navzájem nejbližší body z množiny n bodů.

- Jde o jeden z problémů **výpočetní geometrie**.
- Body mohou ležet na rovině nebo obecně v nějakém mnohodimenzionálním prostoru.
- Body mohou reprezentovat objekty reálného světa nebo záznamy v databázi, texty...
- Příklady aplikací:
  - Bezpečnost letového provozu – hledáme dvě nejbližší letadla ve vzdušném prostoru.
  - Shlukování – hierarchické shlukovací algoritmy postupně spojují sobě nejbližší shluky do jednoho, většího shluku.

Problém nejbližší dvojice bodů – předpoklady

Předpokládejme množinu n bodů $\{P_1, \ldots, P_n\}$, každému bodu P_i odpovídá vektor $\vec{p}_i$ se složkami

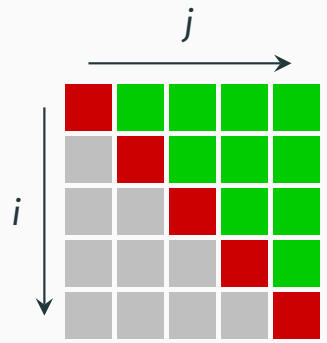
$$\vec{p}_i = (x_i, y_i)$$

v obvyklých kartézských souřadnicích.

Vzdálenost bodů $\vec{p}_i$ a $\vec{p}_j$ budeme počítat pomocí Euklidovské vzdálenosti

$$d(\vec{p}_i, \vec{p}_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Problém nejbližší dvojice bodů – řešení hrubou silou



- Vypočteme vzdálenost všech dvojic bodů $\vec{p}_i$ a $\vec{p}_j$ a nalezneme minimum.
- Stačí počítat jen dvojice bodů pro $j = i + 1, \ldots n$.

Vstup : Množina bodů $\{\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n\}$
Výstup: Vzdálenost dvou nejbližších bodů

```
1 MinDist ← ∞;
2 for i ← 1 to n − 1 do
3 |   for j ← i + 1 to n do
4 |   |   MinDist ← min(MinDist, sqrt((x_i − x_j)^2 + (y_i − y_j)^2));
5 |   end
6 end
7 return MinDist;
```

Problém nejbližší dvojice bodů – řešení hrubou silou, složitost

1. Velikost vstupu – počet bodů n
2. Základní operace
   - Výpočet odmocniny – nejde o triviální záležitost[1].
   - Výpočtu odmocniny se lze vyhnout – jde o rostoucí funkci, lze hledat minimum „čtverců“ vzdáleností.
   - Za základní operaci vezmeme umocňování rozdílů souřadnic.
3. Počet základních operací závisí pouze na n – nejhorší, nejlepší a průměrný případ splývají.

Problém nejbližší dvojice bodů – řešení hrubou silou, složitost (pokrač.)

4. Sestavení vztahů

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i) \\ &= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2) \end{aligned}$$

5. Odstranění odmocniny – snížení složitosti o konstantní faktor, nedošlo k zlepšení asymptotické složitosti, stále je to $\Theta(n^2)$ algoritmus.
6. Později si ukážeme algoritmus s lineárně logaritmickou složitostí.

[1] *https://en.wikipedia.org/wiki/Methods_of_computing_square_roots*

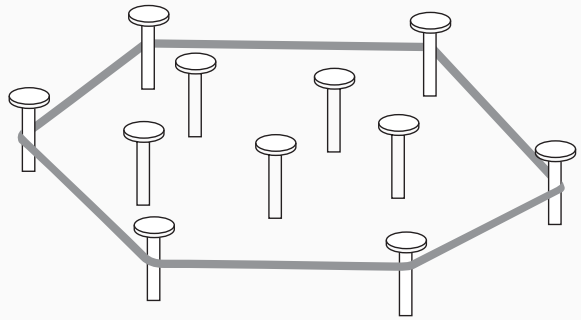
Strategie řešení problémů hrubou silou a úplným prohledáváním

Konvexní obal množiny

Konvexní obal – Convex Hull

Zadání problému

Úkolem je najít konvexní obal (obálku) množiny bodů v prostoru.



- Jde o jeden z problémů **výpočetní geometrie**.
- Body mohou ležet na rovině nebo obecně v nějakém mnohodimenzionálním prostoru.
- Příklady aplikací:
  - Detekce kolizí – počítačová grafika, autonomní vozidla,
  - GIS – bodové senzory, vytvoření oblasti z těchto dat,
  - Optimalizační úlohy – vrcholy konvexního obalu jsou jistým způsobem extrémní; konvexní mnohoúhelník vzniká jako průnik konečného množství polorovin; polorovina je definována nerovnicí…

Konvexní množina bodů

Definice

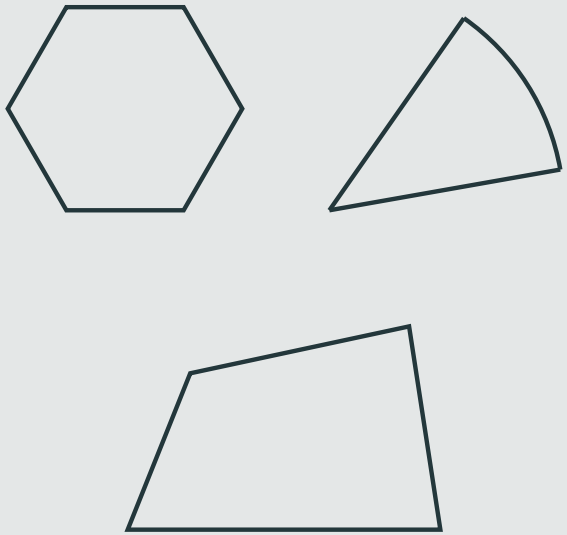
Množina bodů M v rovině se nazývá **konvexní**, jestliže pro libovolnou dvojici bodů $p, q \in M$ úsečka spojující body p a q náleží do množiny M.

Množina, které není konvexní se nazývá **nekonvexní**.

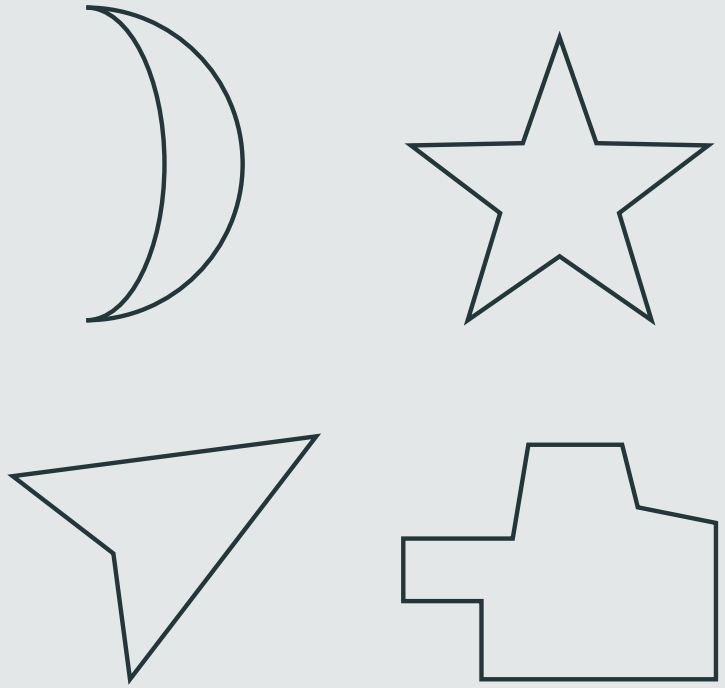
Představíme-li si hranici množiny jako neprůhlednou, znamená konvexita množiny názorně to, že z každého jejího bodu je vidět každý její bod.

Konvexní množina bodů – příklady

Konvexní útvary



Nekonvexní útvary



Konvexní obal

Definice

Konvexním obalem množiny bodů M nazýváme nejmenší konvexní množinu, která obsahuje M.

Výraz „nejmenší“ znamená, že konvexní obal množiny M musí být podmnožinou jakékoliv jiné konvexní podmnožiny obsahující množinu M.

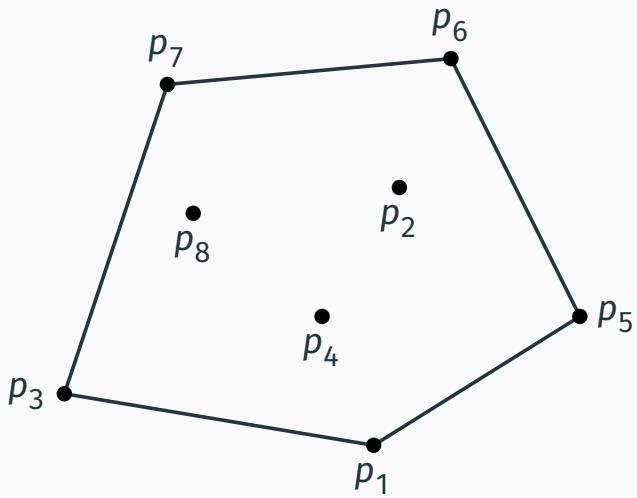
Konvexní obal

- dvouprvkové množiny – úsečka spojující oba body.
- tříprvkové množiny – trojúhelník, pokud body neleží na přímce, jinak je to úsečka spojující nejvzdálenější body.

Konvexní obal (pokrač.)

Věta

*Konvexní obal množiny bodů **M** s více než dvěma body, které neleží na jedné přímce, je konvexní mnohoúhelník, jehož vrcholy náleží do **M**.*

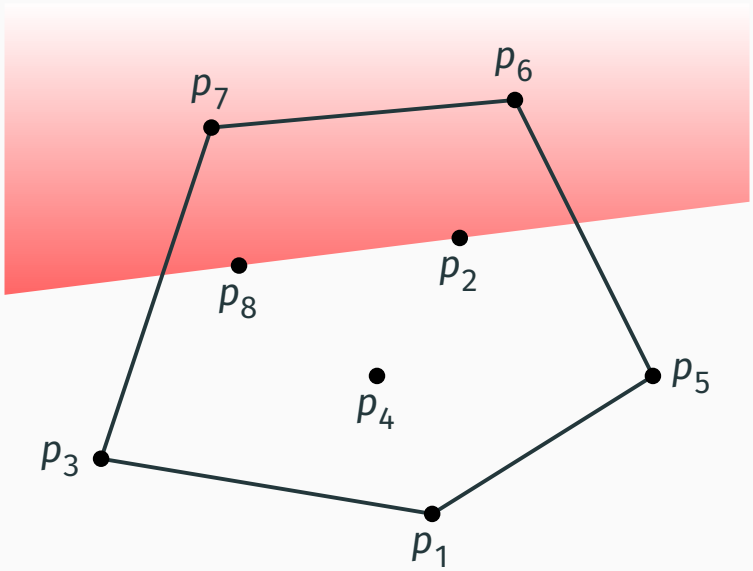


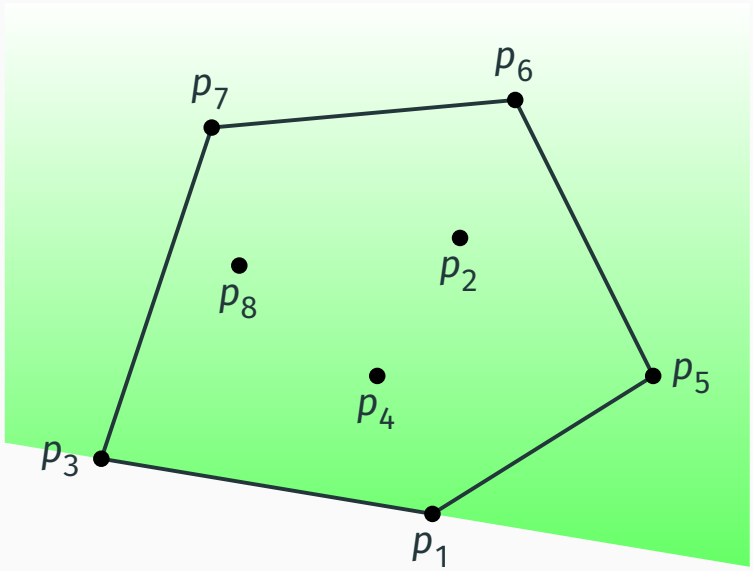
Body množiny ***M***, které specifikují konvexní obal ***M*** se nazývají **extrémní body**.

Problém nalezení konvexního obalu množiny ***M*** redukujeme na nalezení extrémních bodů.

Konvexní obal – řešení hrubou silou

Úsečka $\vec{p}_i\vec{p}_j$ náleží do konvexního obalu množiny M, právě když všechny ostatní body z M leží v jedné z polorovin definovaných přímkou $\vec{p}_i\vec{p}_j$.





Konvexní obal – řešení hrubou silou (pokrač.)

Obecnou rovnici přímky procházející body $\vec{p}_i$ a $\vec{p}_j$ lze psát jako

$$ax + by + c = 0,$$

kde

$$\begin{aligned} a &= y_j - y_i \\ b &= x_i - x_j \\ c &= y_i x_j - x_i y_j \end{aligned}$$

Konvexní obal – řešení hrubou silou (pokrač.)

Přímka definuje dvě poloroviny:

$$ax + by + c < 0 \tag{14}$$

$$ax + by + c > 0 \tag{15}$$

Stačí tedy ověřit, zda pro zbývajících $n - 2$ bodů platí buď nerovnice (14) nebo (15).

Konvexní obal – analýza složitosti

- Musíme prověřit všech $\frac{1}{2}n(n-1)$ dvojic bodů a současně
- pro každou přímku definovanou jednou dvojicí bodů musíme ověřit platnost nerovnic (14) a (15) pro zbývajících $n-2$ bodů.
- Celkově tedy $\left[\frac{1}{2}n(n-1)\right](n-2) \in O(n^3)$.

Strategie řešení problémů hrubou silou a úplným prohledáváním

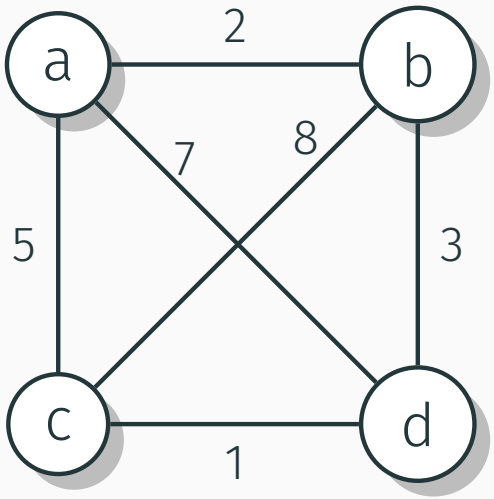
Úplné prohledávání

Úplné prohledávání (Exhaustive search)

- Součást řešení mnoha problémů – nalezení **jednoho prvku**, s nějakou **specifickou vlastností**, z množiny prvků, tzv. domény, která **exponenciálně** nebo rychleji roste s velikostí problému.
- Hledaný element je typicky **kombinatorické povahy** – permutace, kombinace, podmnožina.
- Typicky jde o **optimalizační úlohy** – typicky hledáme maximum, minimum. Například minimalizujeme délku cesty, maximalizujeme zisk.

Úplné prohledávání je strategie řešení problému hrubou silou spočívající v otestování všech prvků uvažované domény.

Problém obchodního cestujícího – příklad



Trasa	Délka trasy l
$a \to b \to c \to d \to a$	$2 + 8 + 1 + 7 = 18$
$a \to b \to d \to c \to a$	$2 + 3 + 1 + 5 = 11$
$a \to c \to b \to d \to a$	$5 + 8 + 3 + 7 = 23$
$a \to c \to d \to b \to a$	$5 + 1 + 3 + 2 = 11$
$a \to d \to b \to c \to a$	$7 + 3 + 8 + 5 = 23$
$a \to d \to c \to b \to a$	$7 + 1 + 8 + 2 = 18$

Úplné prohledávání

Problém obchodního cestujícího (Traveling Salesman Problem)

Problém batohu

Strategie řešení problémů hrubou silou a úplným prohledáváním

Průchody grafem

Průchod grafem do hloubky a do šířky

Algoritmus průchodu grafem do hloubky

```
   Input  : Graf G(V, E), počáteční vrchol s ∈ V
   Output: DF-strom
 1 Init(V, s);
 2 while stack ≠ ∅ do
 3 |   u ← Top(stack);
 4 |   switch state [u] do
 5 |   |   case discovered do
 6 |   |   |   ProcessDiscoveredVertex(u);
 7 |   |   end
 8 |   |   case current do
 9 |   |   |   ProcessCurrentVertex(u);
10 |   |   end
11 |   end
12 end
```

Algoritmus průchodu grafem do hloubky (pokrač.)

```
1  Procedure Init(V, s)
       Input  : Množina vrcholů V, počáteční vrchol s ∈ V
2      foreach u ∈ V do
3          state [u] ← unknown;
4          d [u] ← f [u] ← ∞;
5          π [u] ← nothing;
6      end
7      state [s] ← discovered;
8      stack ← ∅;
9      Push(stack, s);
10     time ← 0;
11 end
```

Algoritmus průchodu grafem do hloubky (pokrač.)

```
1  Procedure ProcessDiscoveredVertex(u)
   |  Input  : Vrchol u ∈ V
2  |  state [u] ← current;
3  |  d [u] ← time ← time + 1;
4  |  foreach v ∈ Adj(G, u) do
5  |  |  if state [v] = unknown then
6  |  |  |  state [v] ← = discovered;
7  |  |  |  π [v] ← u;
8  |  |  |  Push(stack, v);
9  |  |  end
10 |  end
11 end
```

Algoritmus průchodu grafem do hloubky (pokrač.)

```
1 Procedure ProcessCurrentVertex(u)
  |   Input  : Vrchol u ∈ V
2 |   state[u] ← finished;
3 |   f[u] ← time ← time + 1;
4 |   Pop(stack);
5 end
```

Animace průchodu grafem do hloubky – legenda

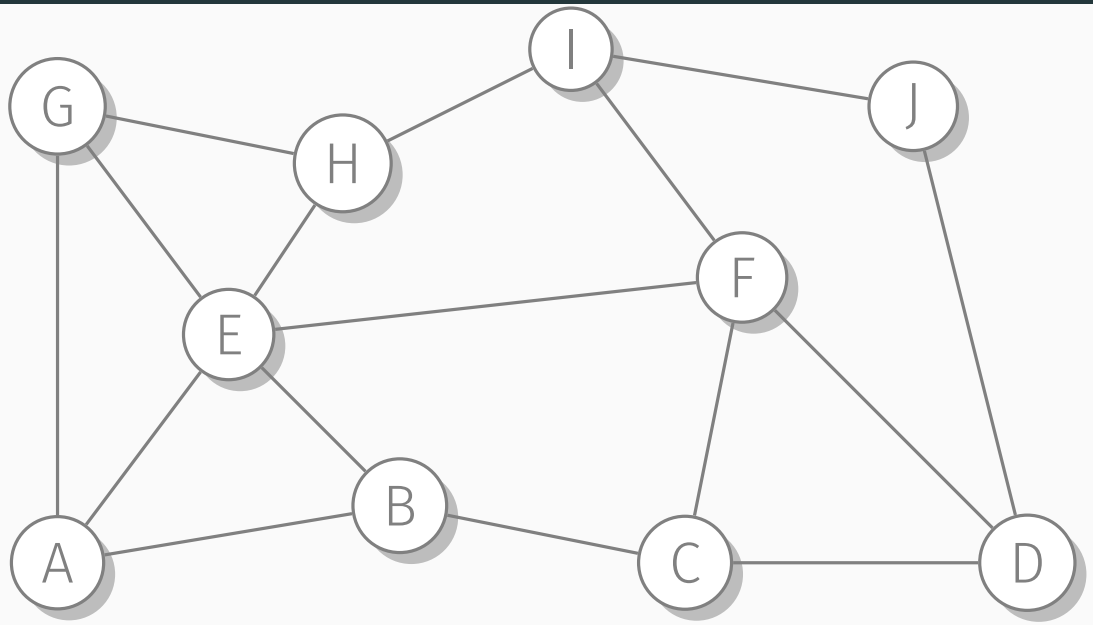
Vrcholy grafu:

šedá	vrchol ve stavu unknown
žlutá	vrchol ve stavu discovered
červená	vrchol ve stavu current
modrá	vrchol ve stavu finished

Hrany grafu:

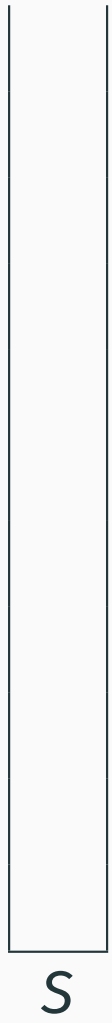
šedá	hrana mezi vrcholy ve stavu unknown nebo hrana nepatřící do DF-stromu
žlutá	hrana incidentní s vrcholy ve stavu discovered
červená	hrana incidentní s vrcholem ve stavu current
modrá	hrana mezi vrcholy ve stavu finished

Průchod grafem do hloubky, krok 1

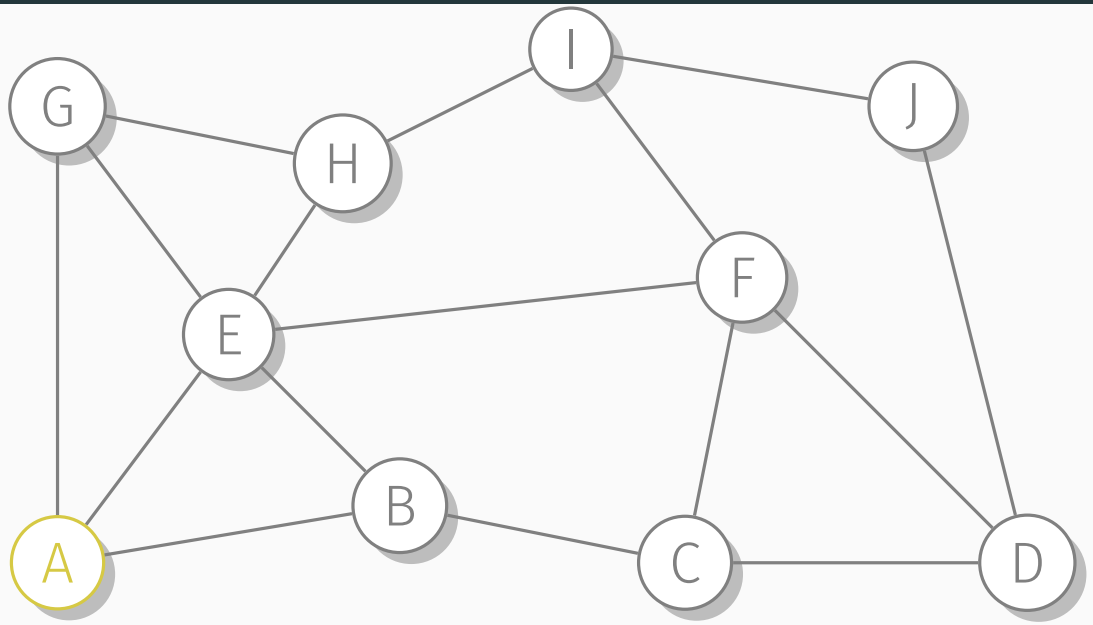


v	$d[v]$	$f[v]$	$\pi[v]$
A	∞	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/



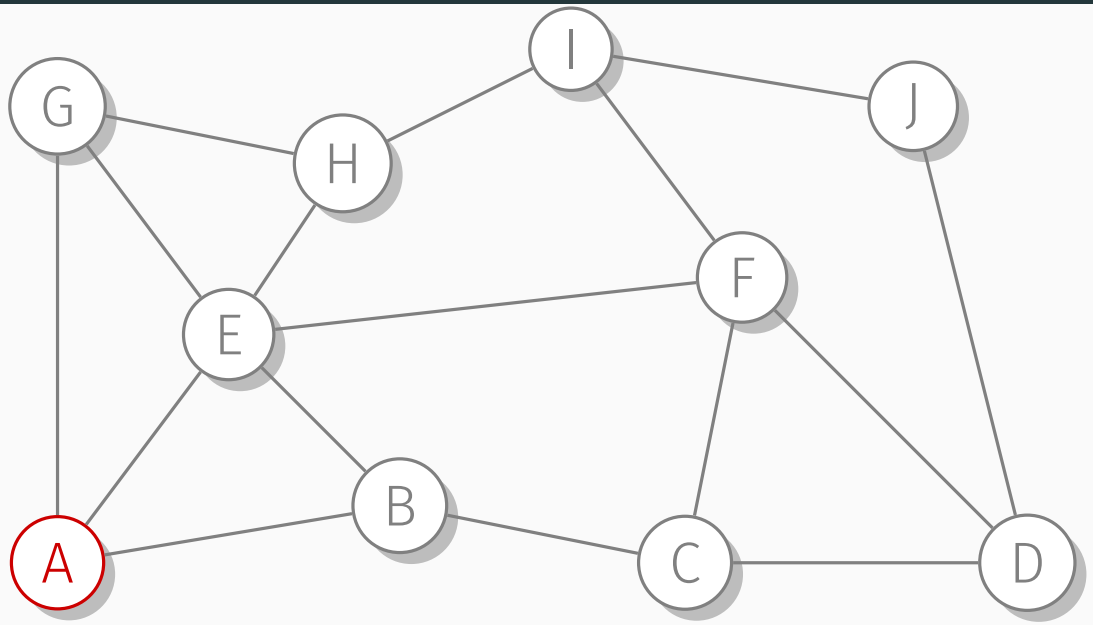
Průchod grafem do hloubky, krok 2



v	$d[v]$	$f[v]$	$\pi[v]$
A	∞	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

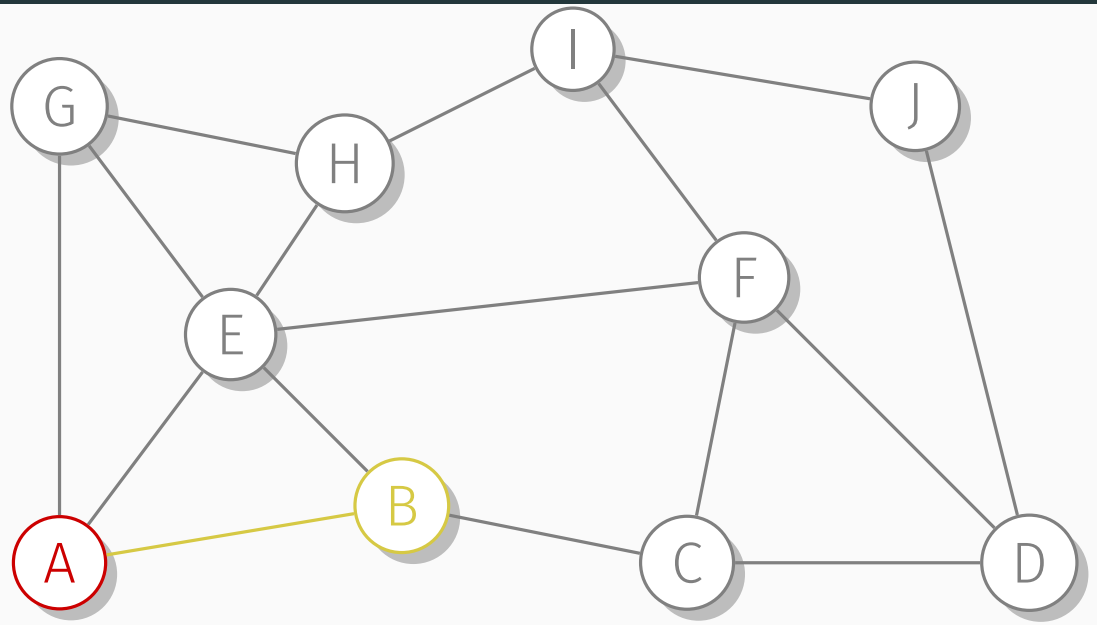
Průchod grafem do hloubky, krok 3



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

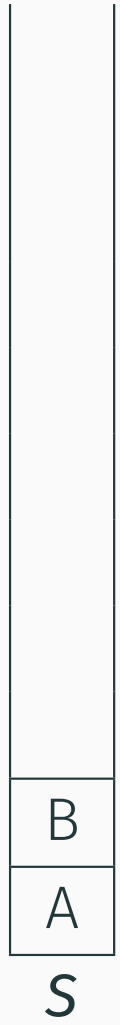
v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

Průchod grafem do hloubky, krok 4

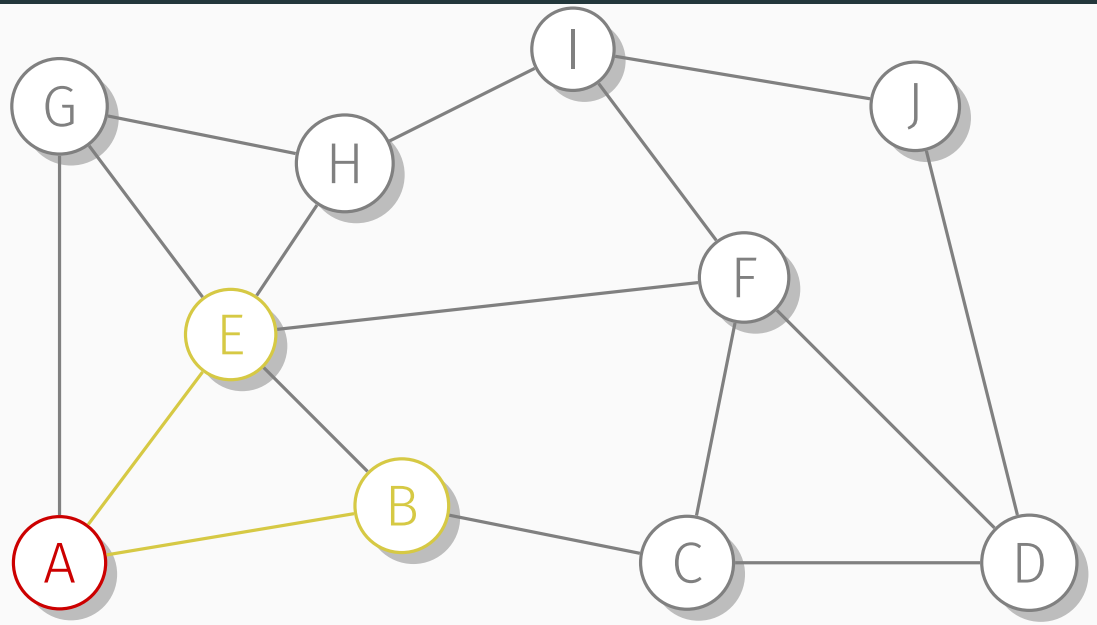


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

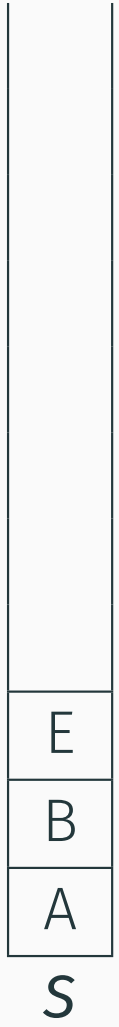


Průchod grafem do hloubky, krok 5

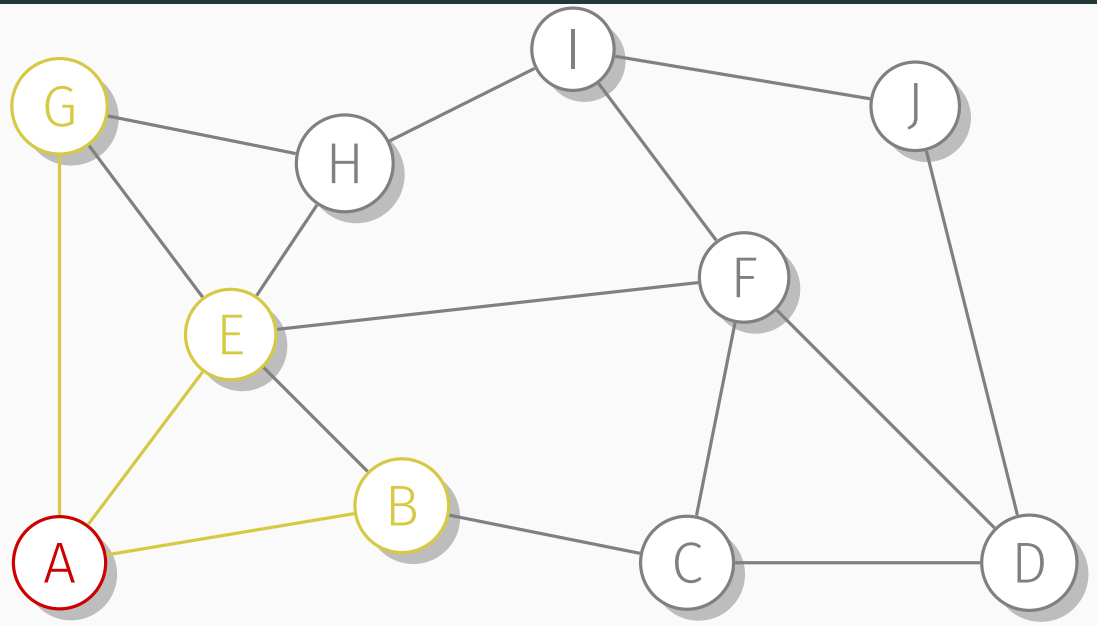


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/



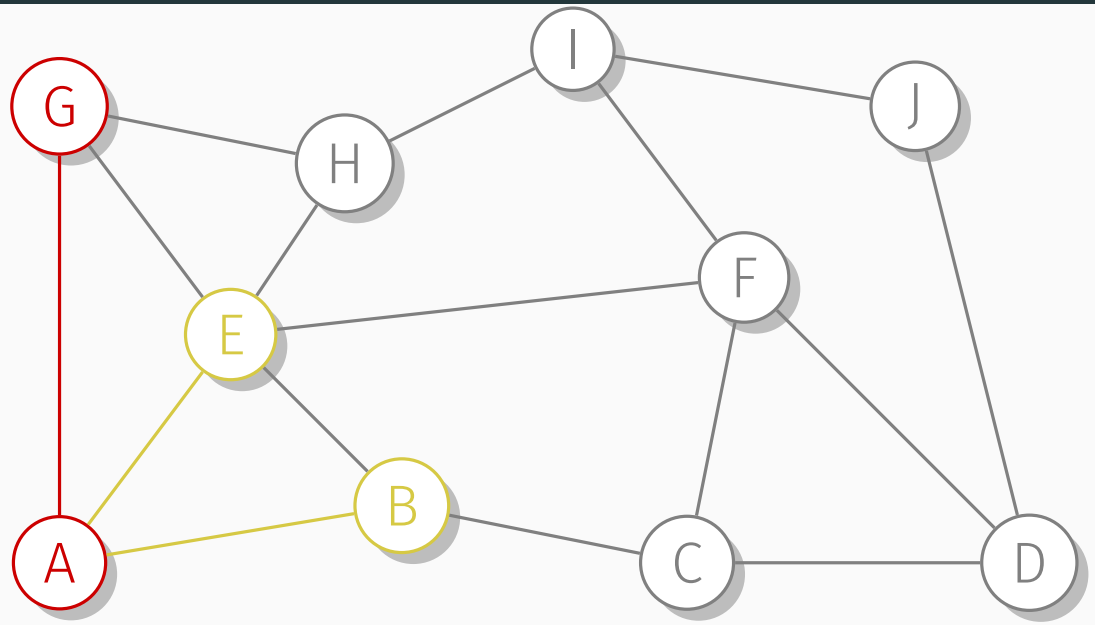
Průchod grafem do hloubky, krok 6



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	A
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

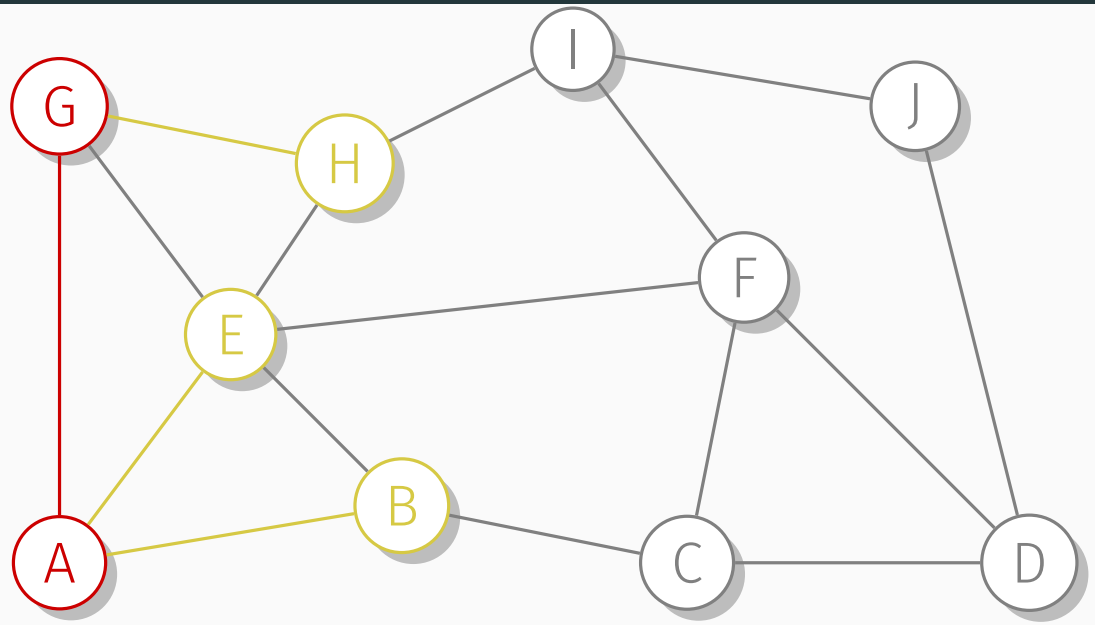
Průchod grafem do hloubky, krok 7



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

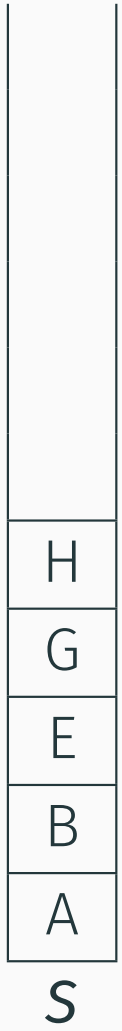
v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

Průchod grafem do hloubky, krok 8

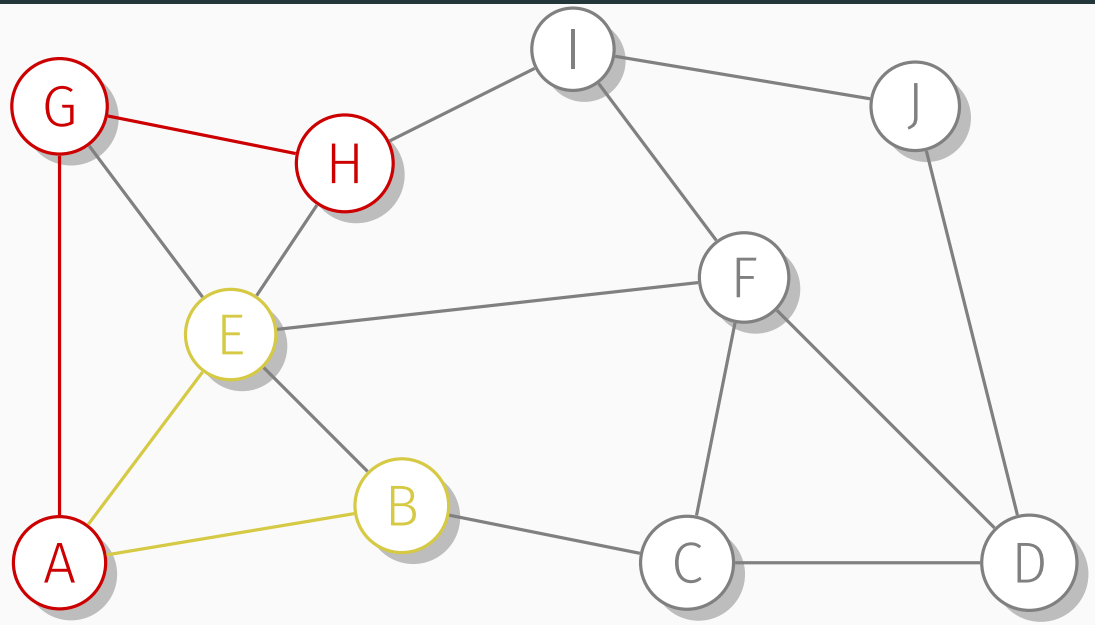


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	∞	∞	G
I	∞	∞	/
J	∞	∞	/

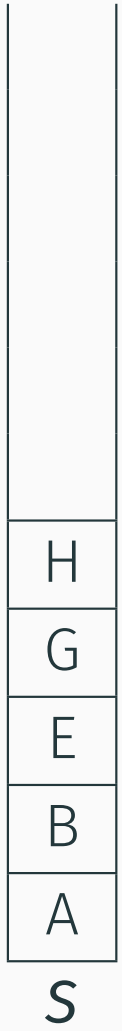


Průchod grafem do hloubky, krok 9

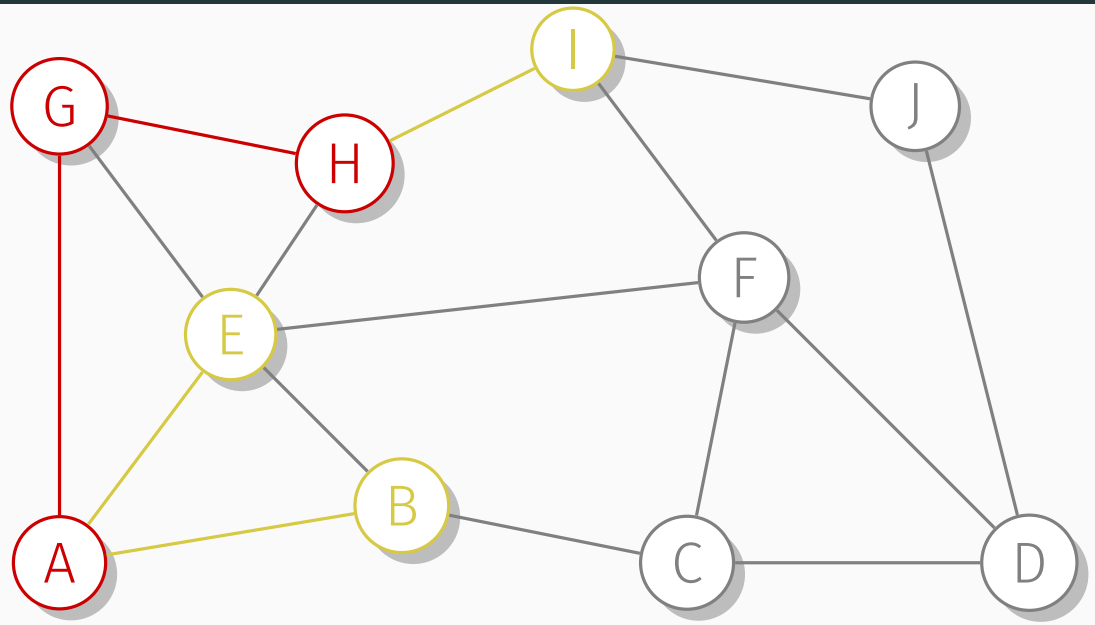


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	3	∞	G
I	∞	∞	/
J	∞	∞	/

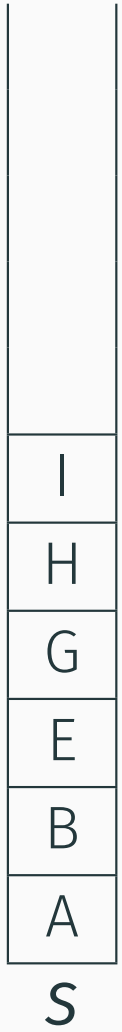


Průchod grafem do hloubky, krok 10

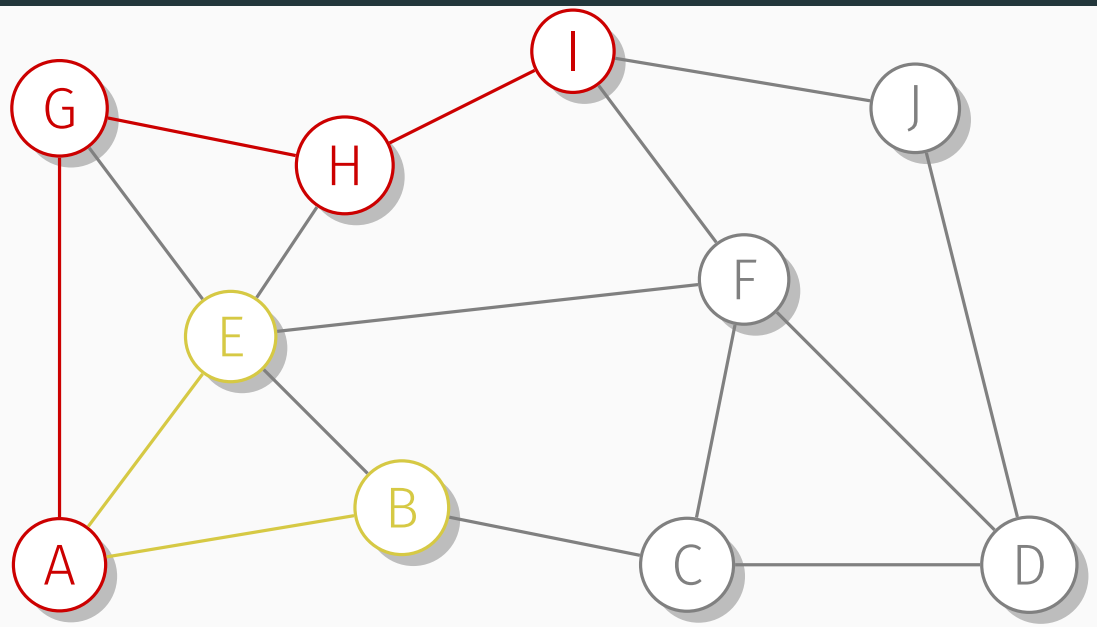


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	3	∞	G
I	∞	∞	H
J	∞	∞	/

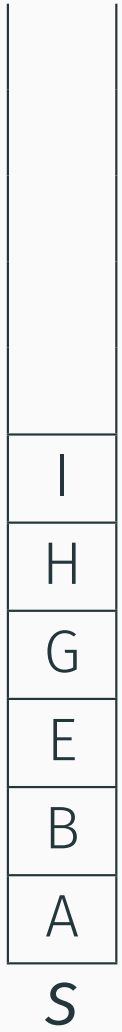


Průchod grafem do hloubky, krok 11

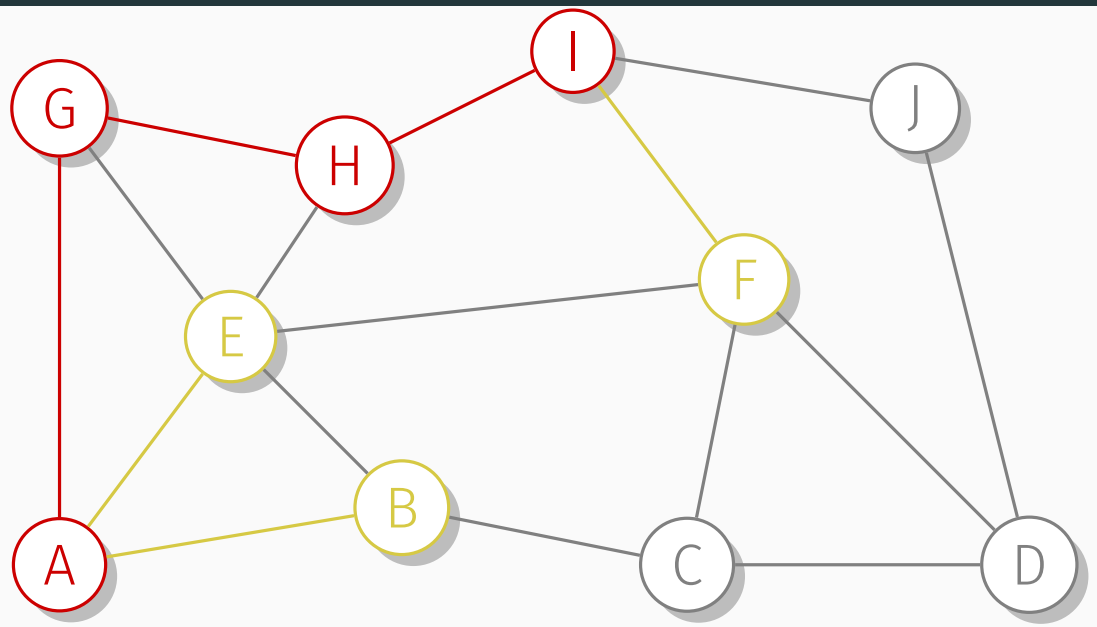


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	∞	∞	/

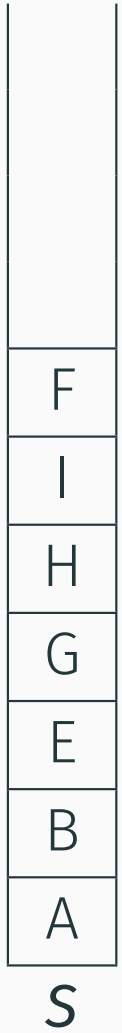


Průchod grafem do hloubky, krok 12

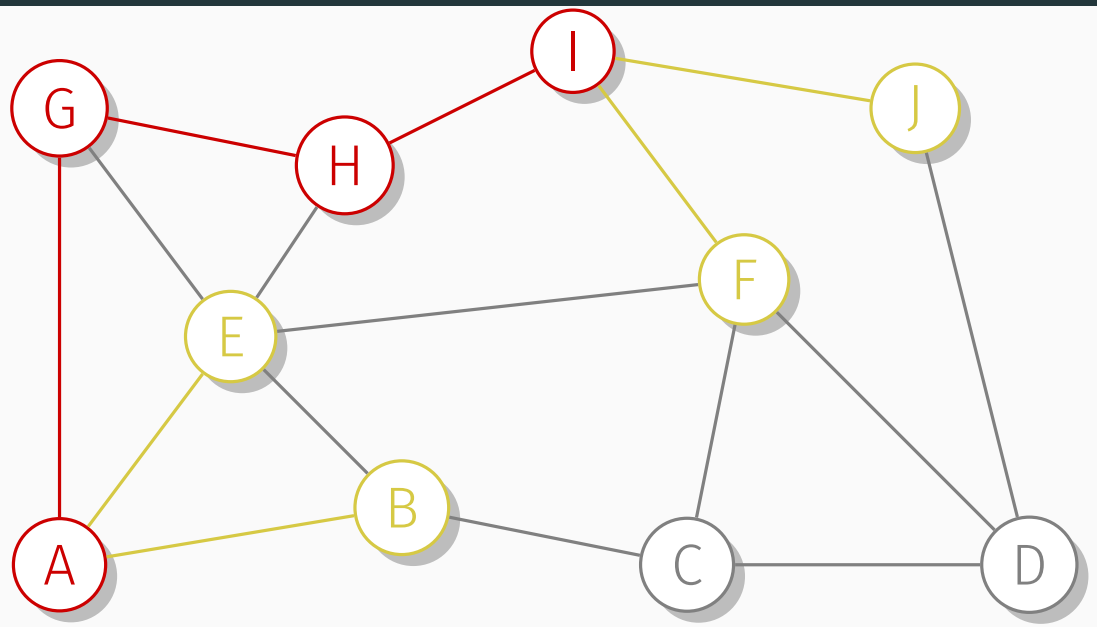


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	∞	∞	/

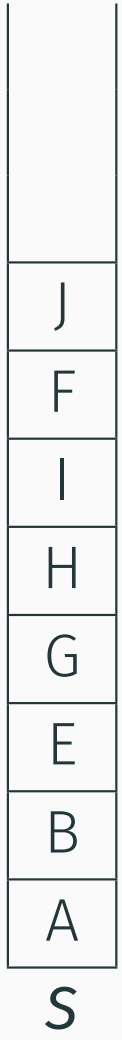


Průchod grafem do hloubky, krok 13

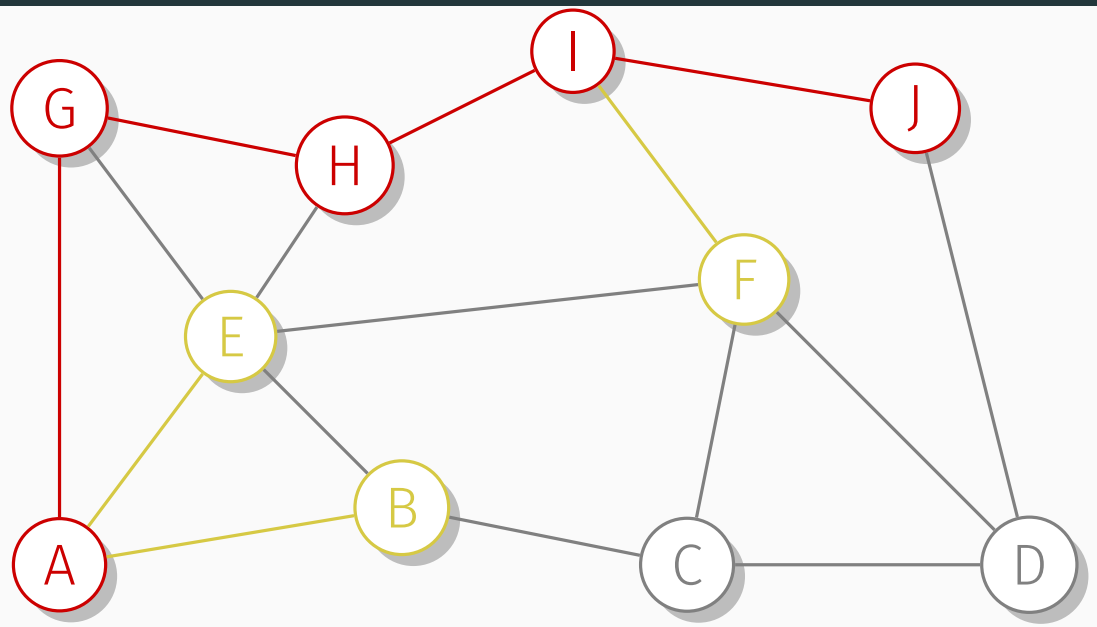


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	∞	∞	I

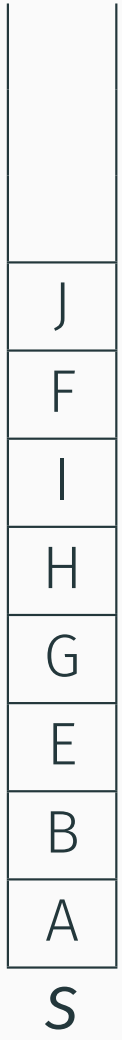


Průchod grafem do hloubky, krok 14

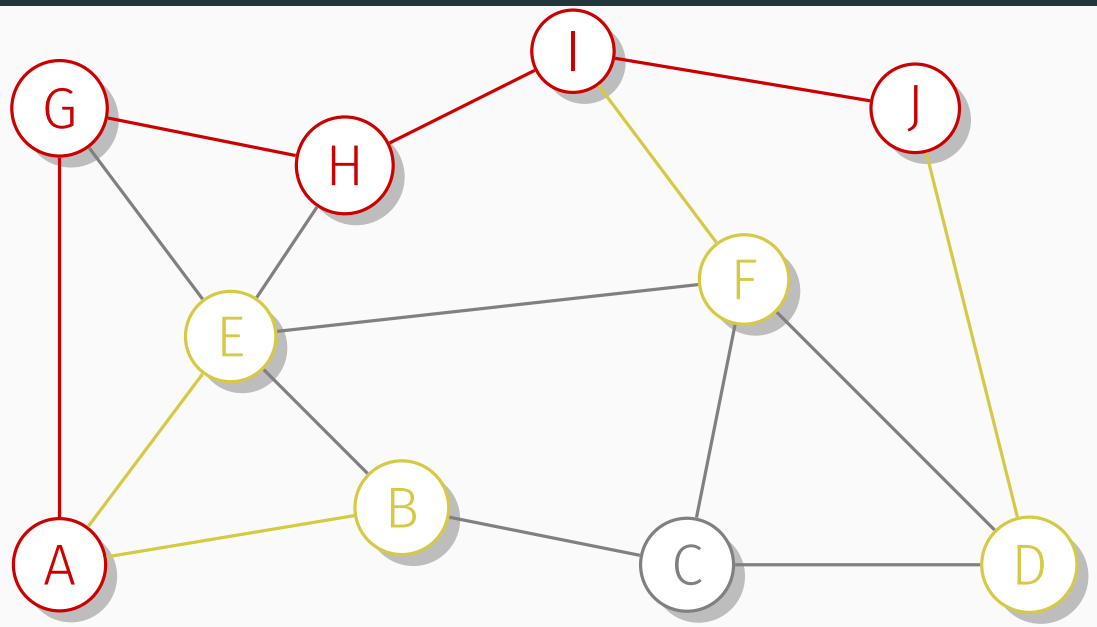


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

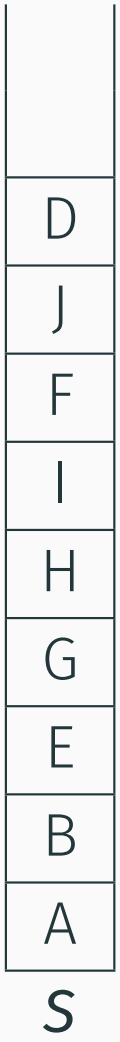


Průchod grafem do hloubky, krok 15

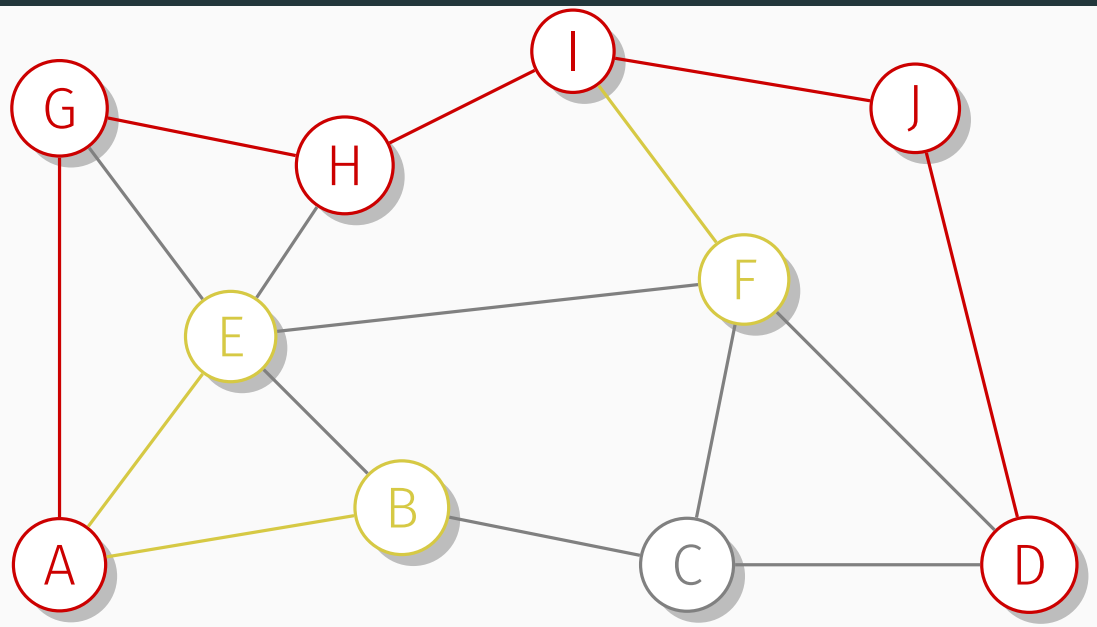


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

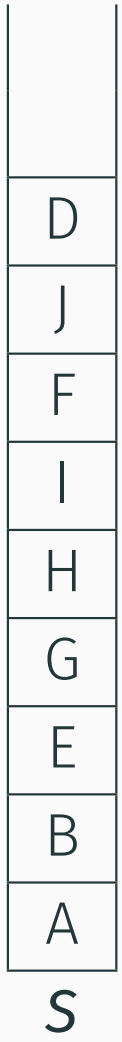


Průchod grafem do hloubky, krok 16

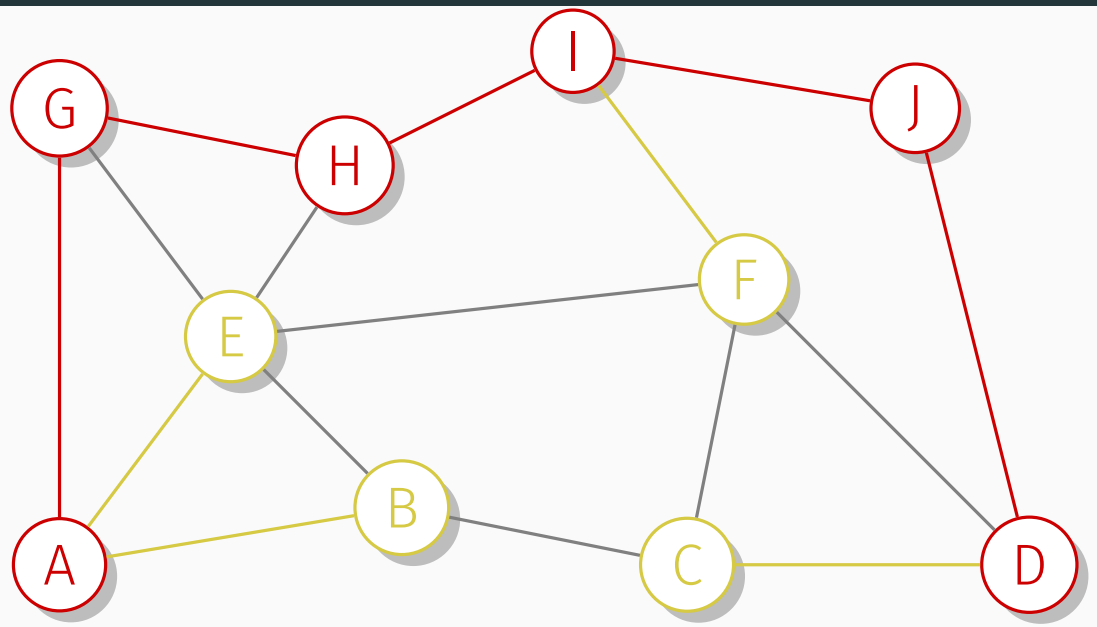


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

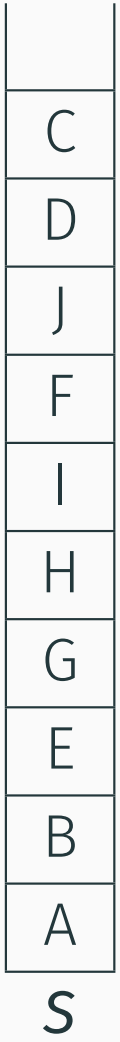


Průchod grafem do hloubky, krok 17

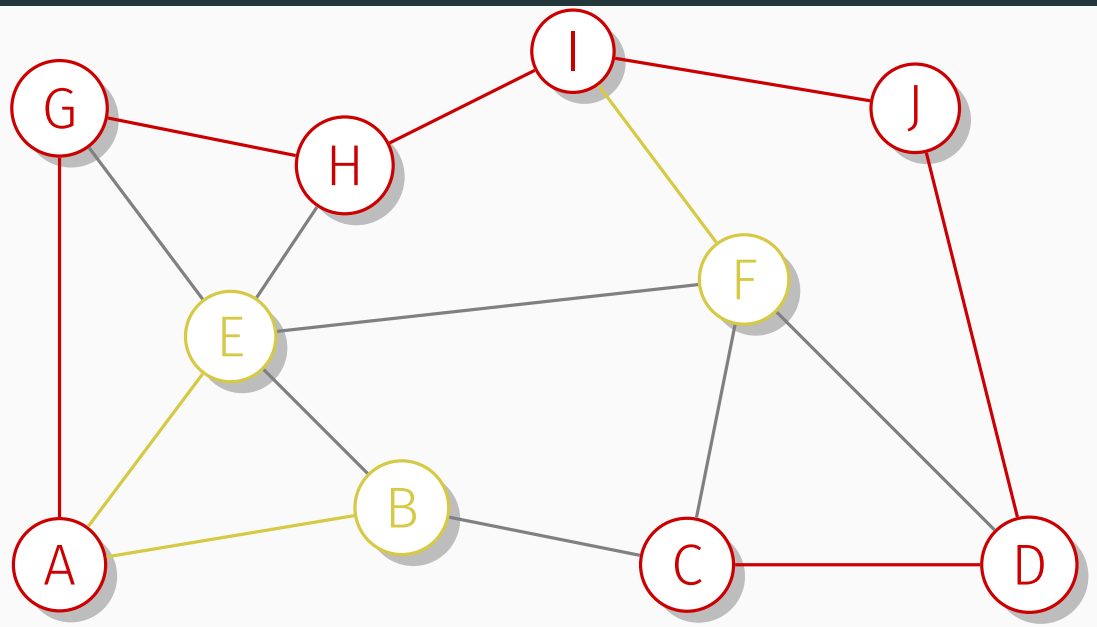


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	D
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

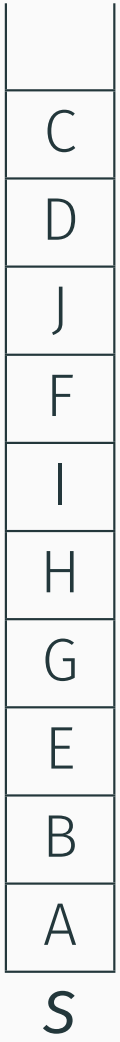


Průchod grafem do hloubky, krok 18

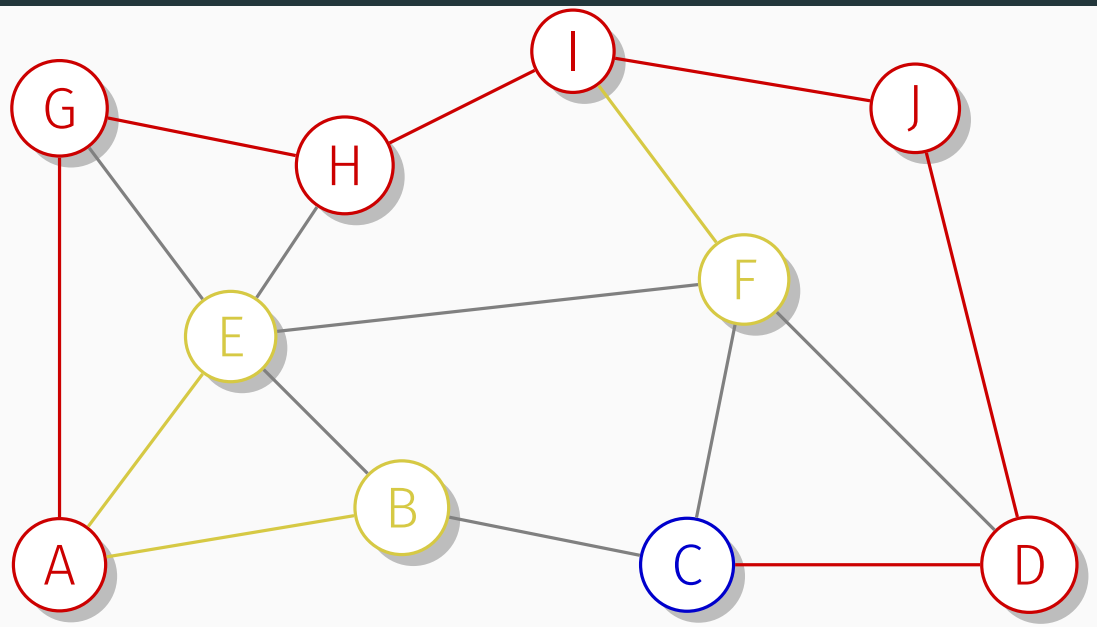


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	∞	D
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

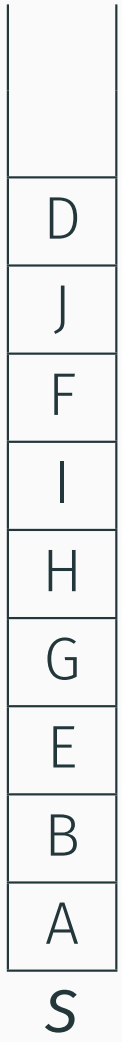


Průchod grafem do hloubky, krok 19

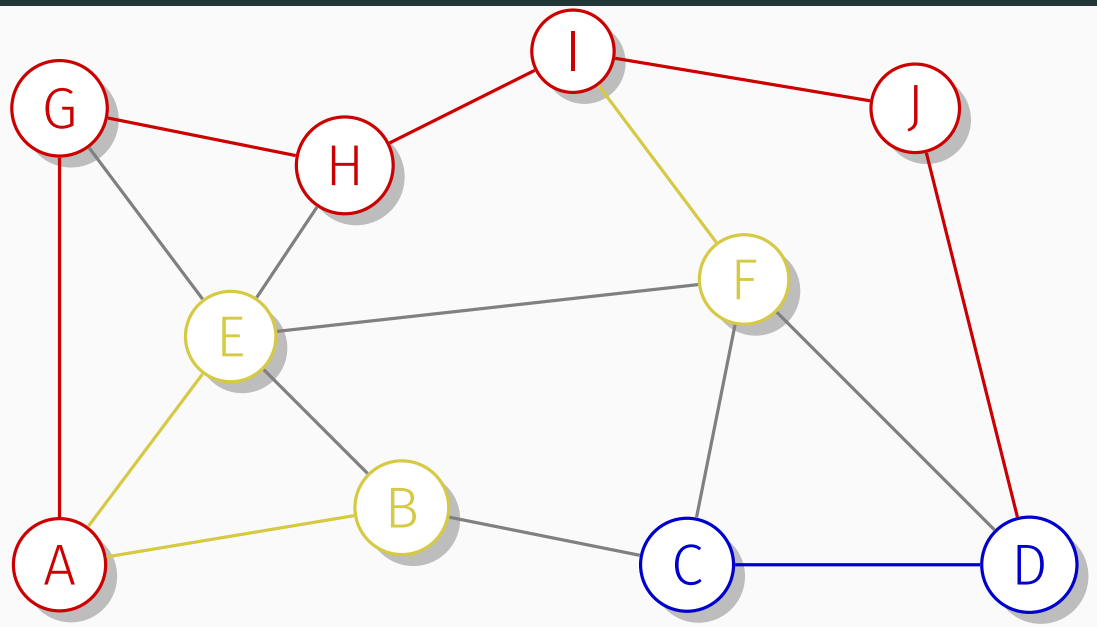


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

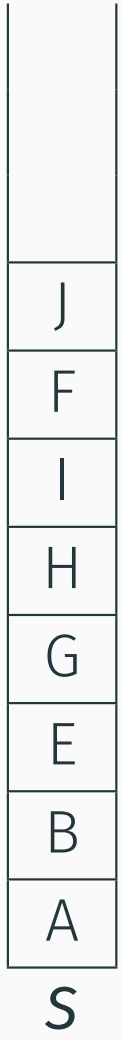


Průchod grafem do hloubky, krok 20

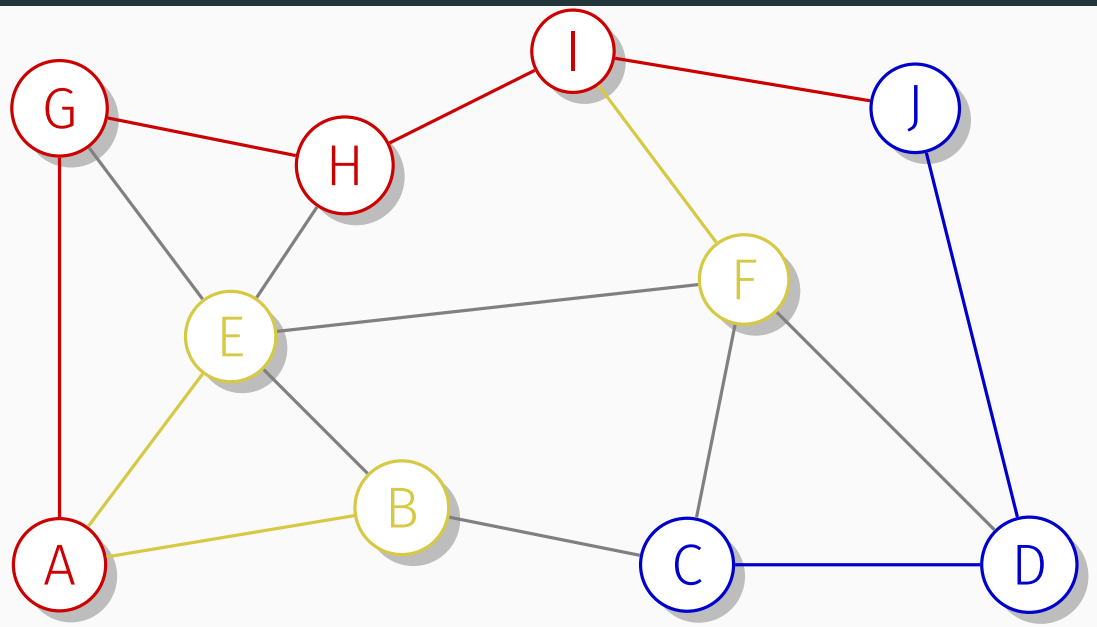


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

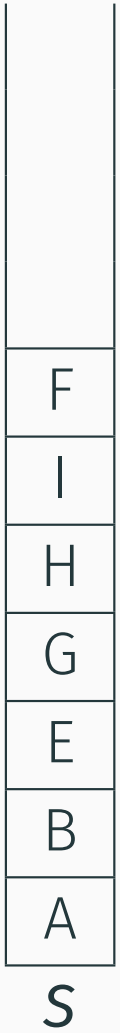


Průchod grafem do hloubky, krok 21

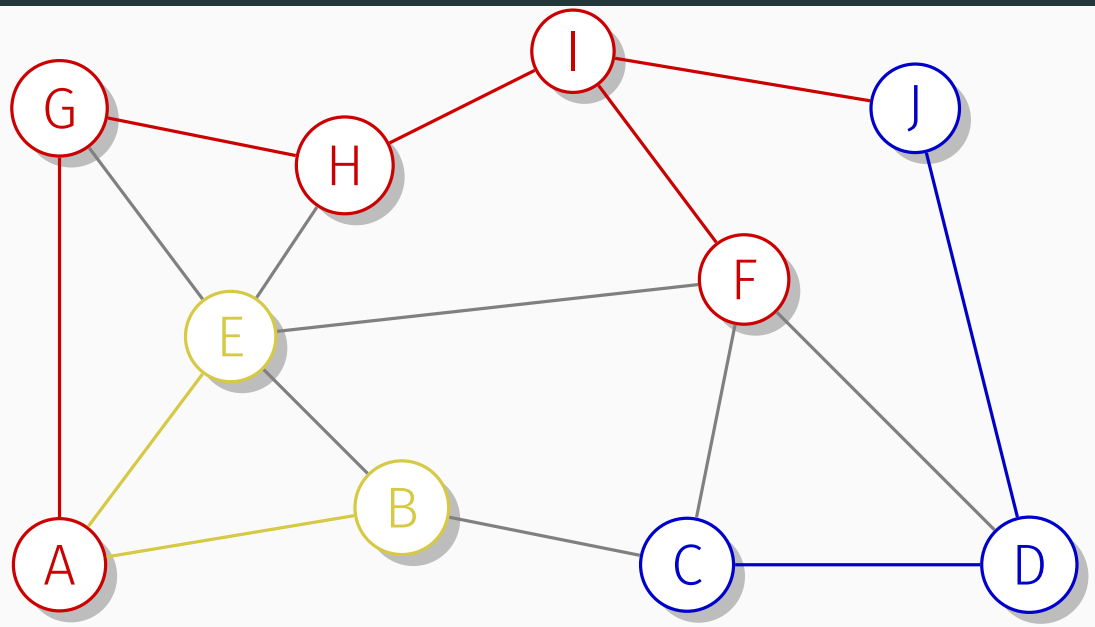


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	10	I

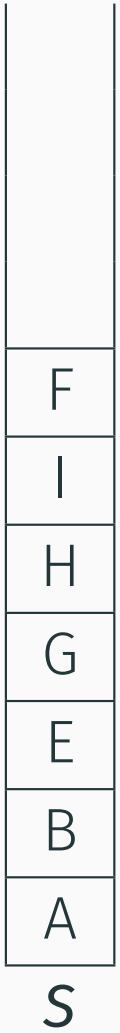


Průchod grafem do hloubky, krok 22

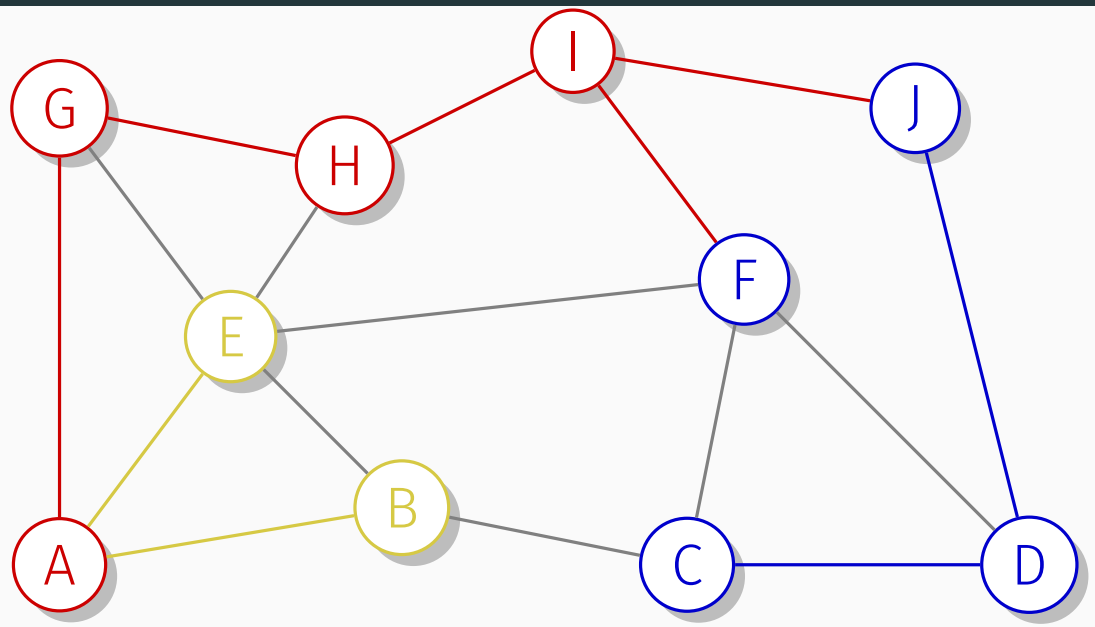


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	10	I

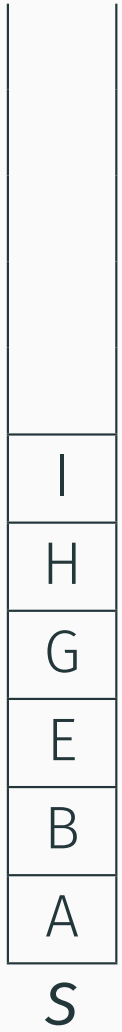


Průchod grafem do hloubky, krok 23

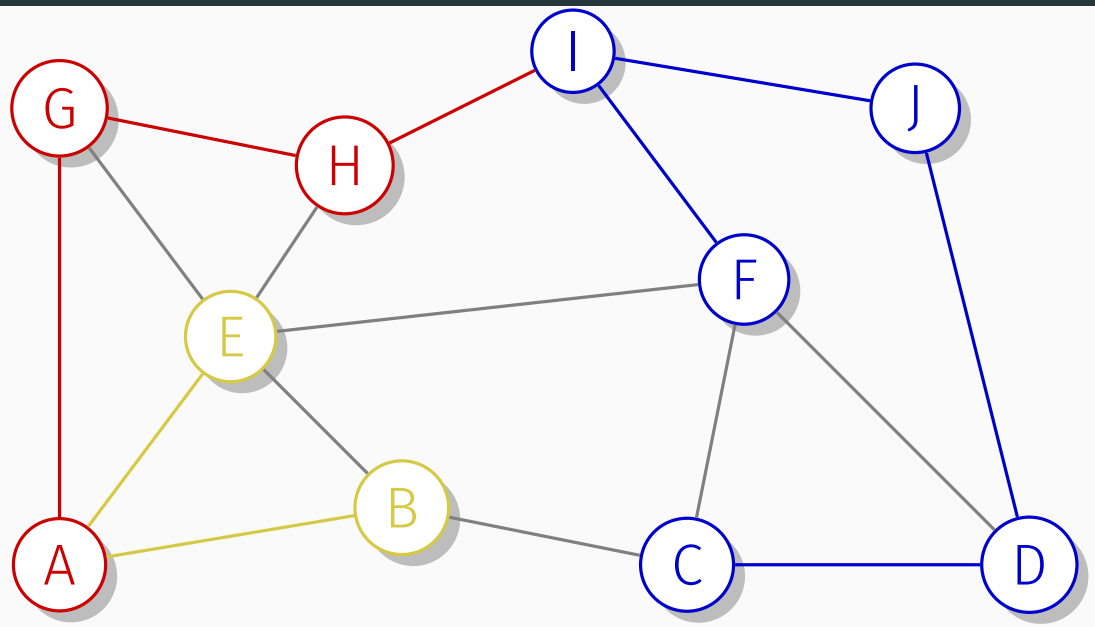


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	10	I

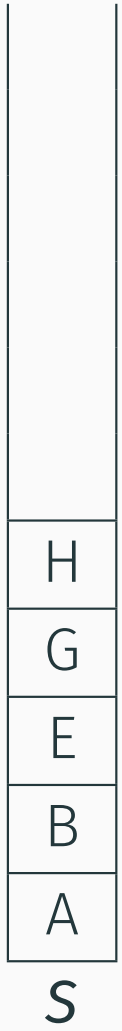


Průchod grafem do hloubky, krok 24

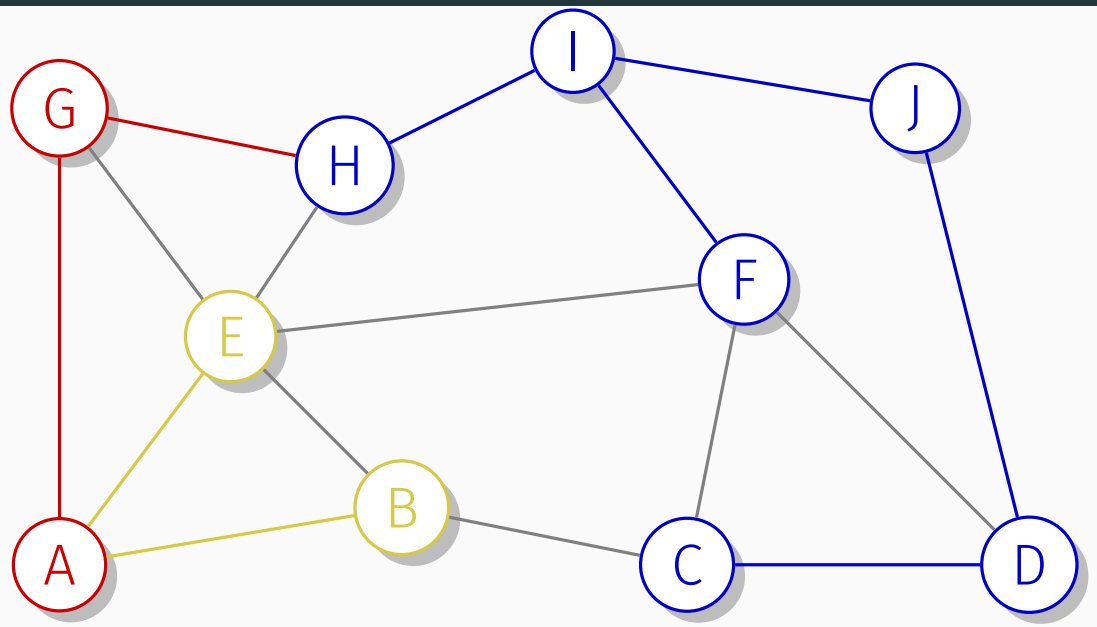


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	∞	A
H	3	∞	G
I	4	13	H
J	5	10	I



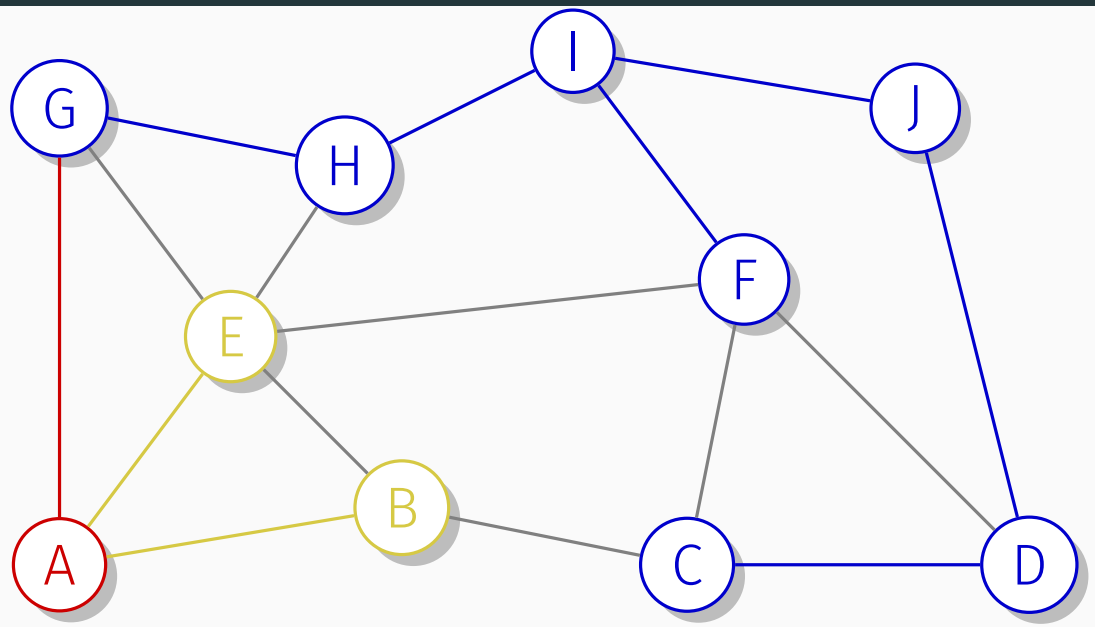
Průchod grafem do hloubky, krok 25



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

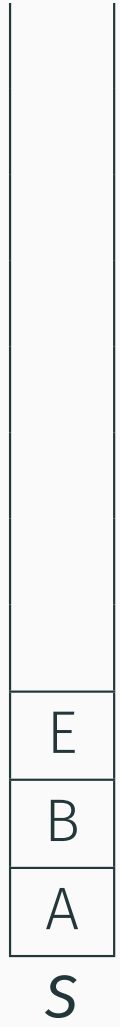
v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	∞	A
H	3	14	G
I	4	13	H
J	5	10	I

Průchod grafem do hloubky, krok 26

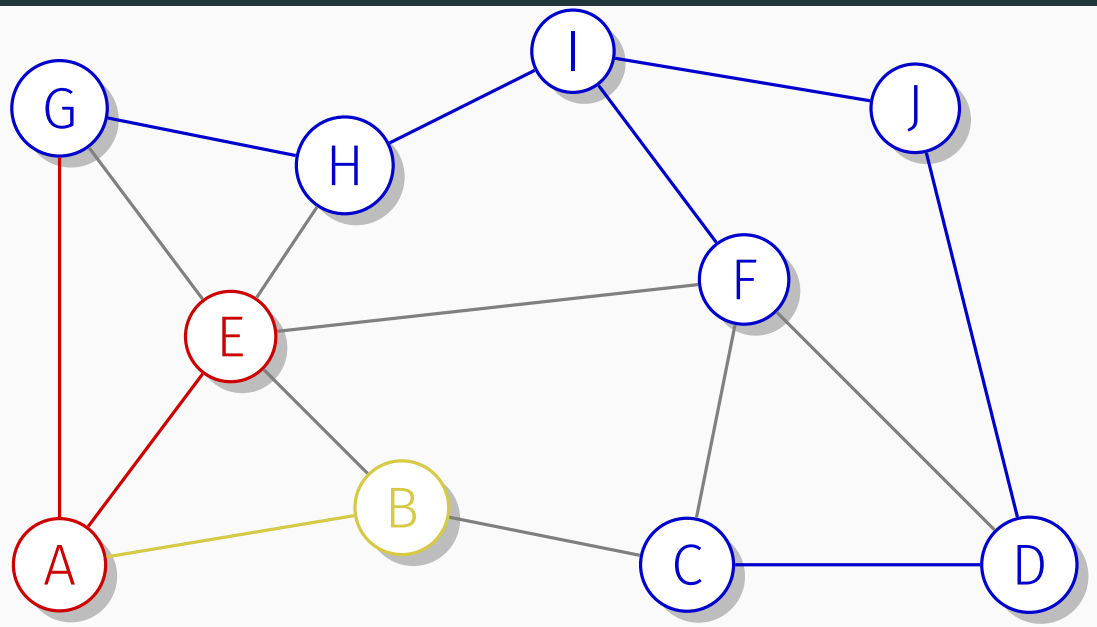


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I



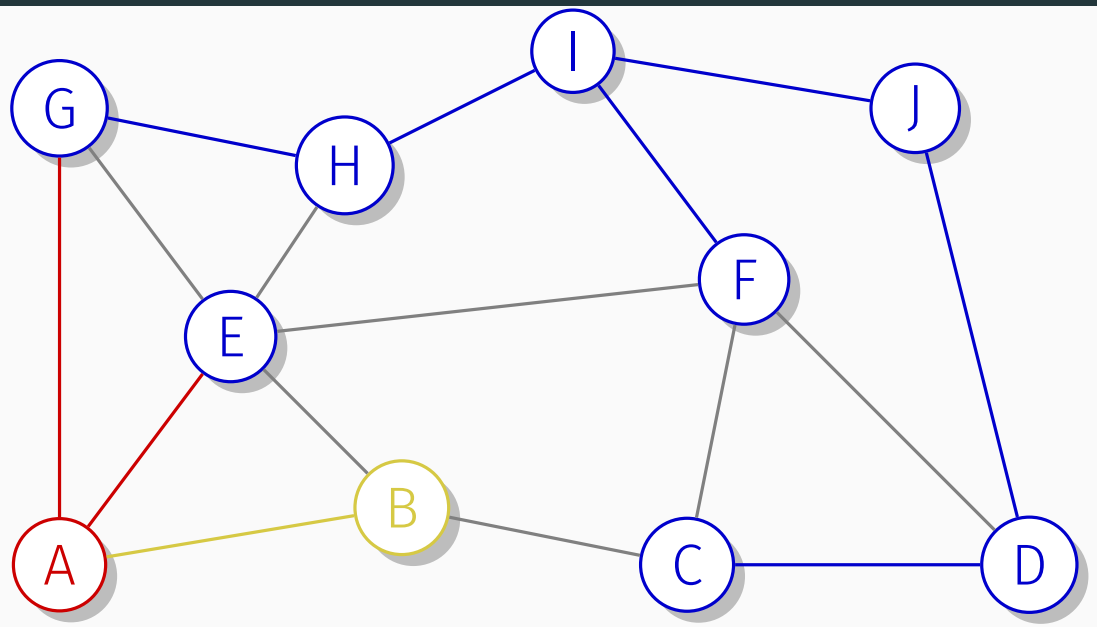
Průchod grafem do hloubky, krok 27



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	16	∞	A

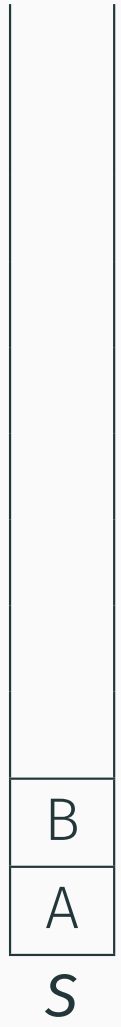
v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

Průchod grafem do hloubky, krok 28

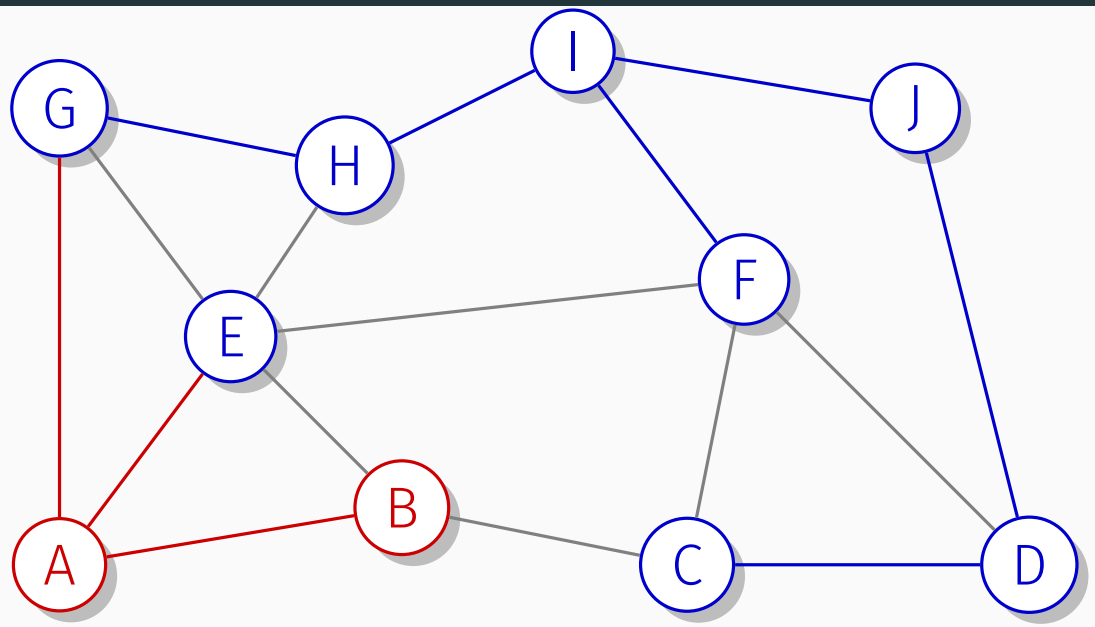


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	16	17	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

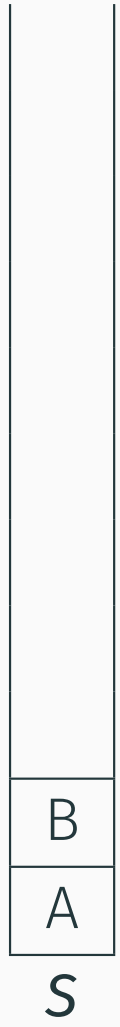


Průchod grafem do hloubky, krok 29

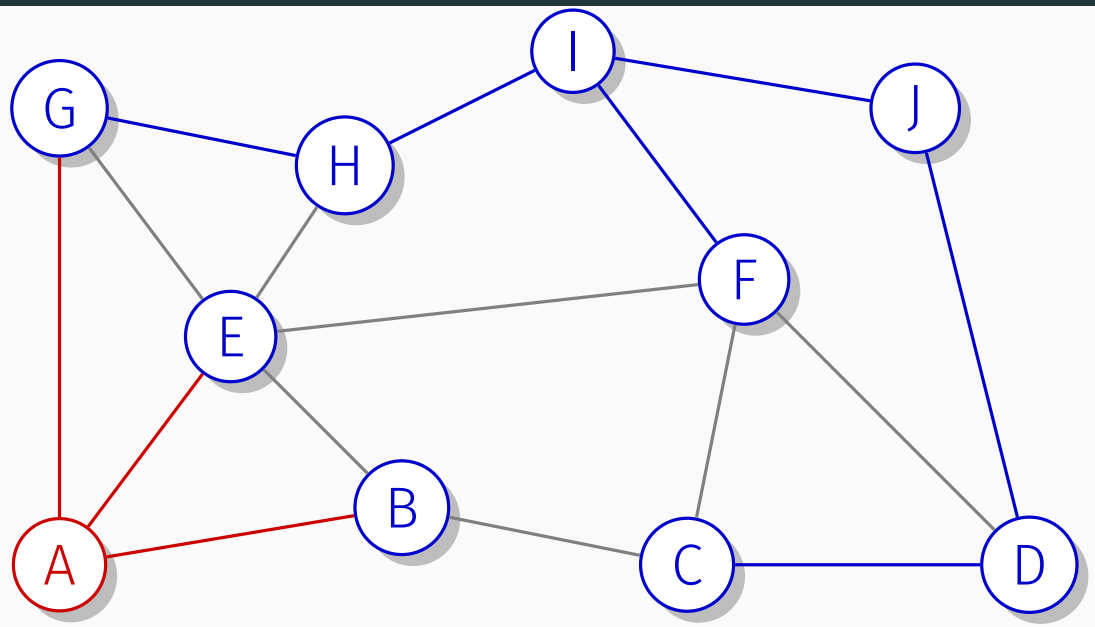


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	18	∞	A
C	7	8	D
D	6	9	J
E	16	17	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I



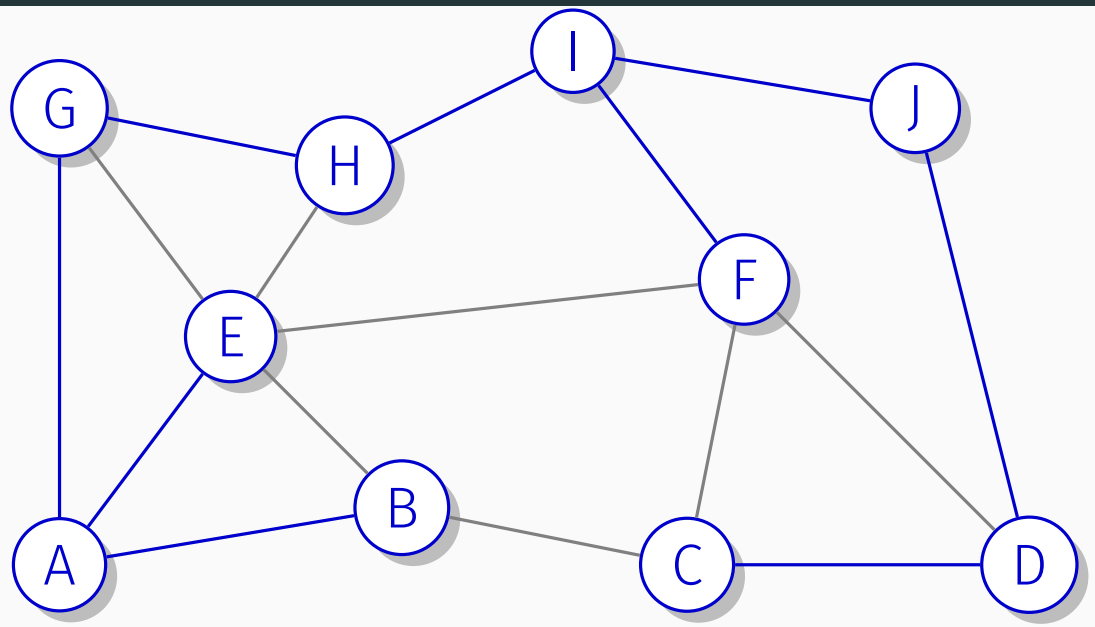
Průchod grafem do hloubky, krok 30



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	18	19	A
C	7	8	D
D	6	9	J
E	16	17	A

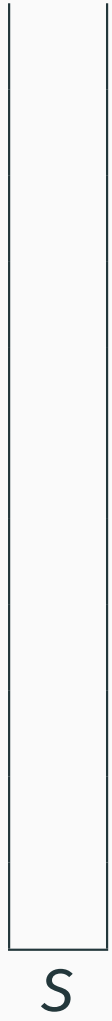
v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

Průchod grafem do hloubky, krok 31



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	20	/
B	18	19	A
C	7	8	D
D	6	9	J
E	16	17	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I



Algoritmus průchodu grafem do šířky

Input : Graf $G(V, E)$, počáteční vrchol $s \in V$
Output: BF-strom

```
1  foreach u ∈ V/{s} do
2  |   state [u] ← unknown;
3  |   d [u] ← ∞;
4  |   π [u] ← nothing;
5  end
6  Q ← ∅;
7  state [s] ← discovered;
8  d [s] ← 0;
9  π [s] ← nothing;
10 Enqueue(Q, s);
```

Algoritmus průchodu grafem do šířky (pokrač.)

```
11 while Q ≠ ∅ do
12 |   u ← Dequeue(Q);
13 |   foreach v ∈ Adj(G, u) do
14 |   |   if state [v] = unknown then
15 |   |   |   state [v] ← discovered;
16 |   |   |   d [v] ← d [u] + 1;
17 |   |   |   π [v] ← u;
18 |   |   |   Enqueue (Q,v);
19 |   |   end
20 |   end
21 |   state [u] ← finished;
22 end
```

Animace průchodu grafem do šířky – legenda

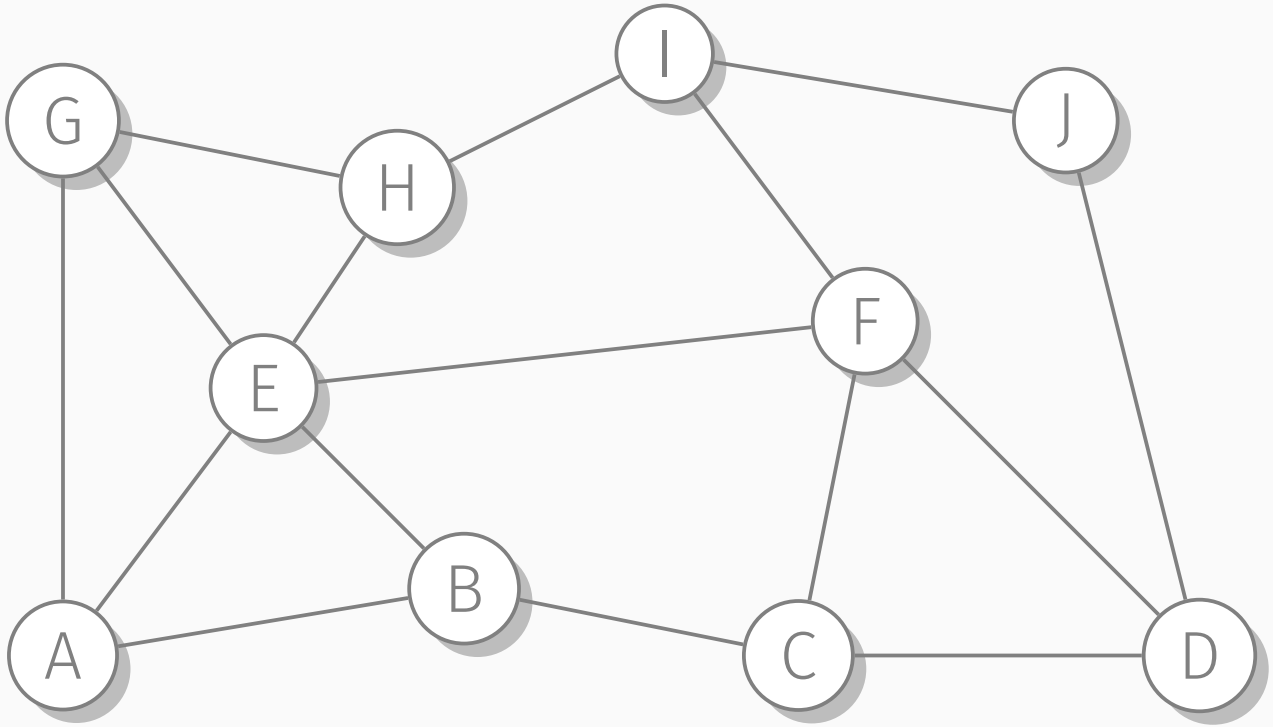
Vrcholy grafu

šedá	vrchol ve stavu unknown
žlutá	vrchol ve stavu discovered
červená	právě zpracovávaný vrchol
modrá	vrchol ve stavu finished

Hrany grafu

šedá	hrana mezi vrcholy ve stavu unknown nebo hrana nepatřící do BF-stromu
žlutá	hrana incidentní s vrcholy ve stavu discovered
červená	hrana incidentní s právě zpracovávaným vrcholem
modrá	hrana mezi vrcholy ve stavu finished

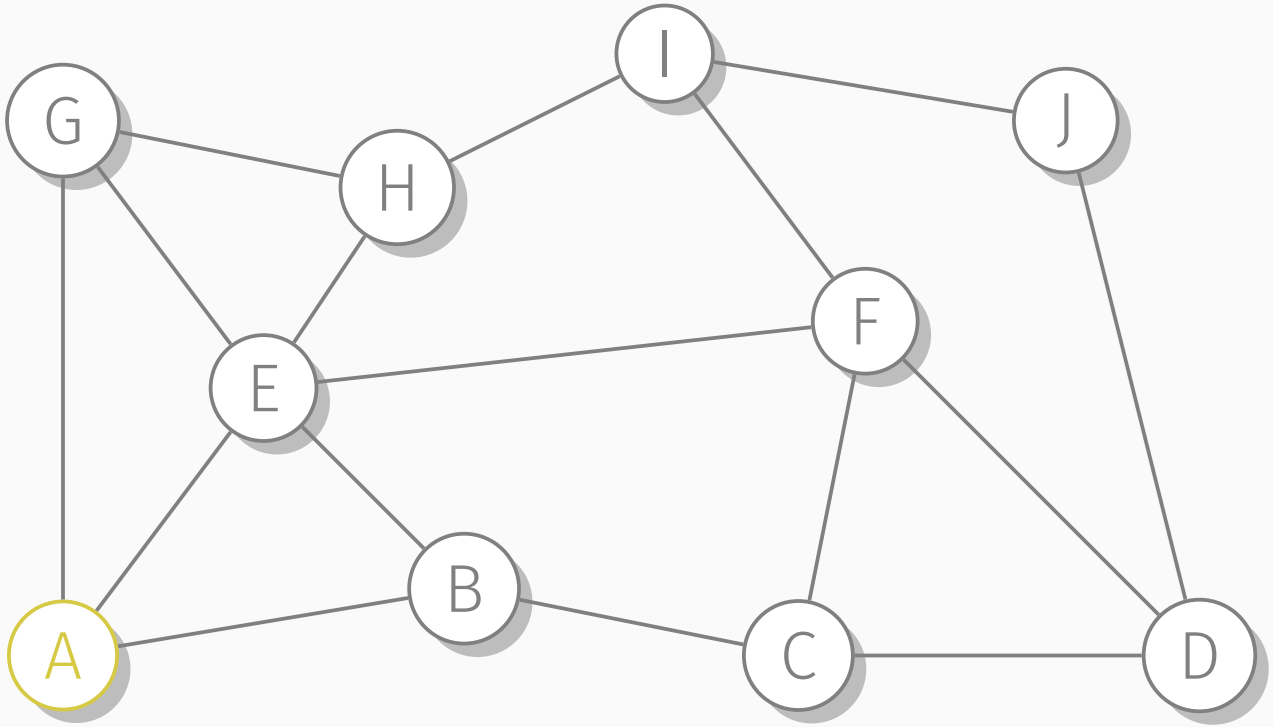
Průchod grafem do šířky, krok 1



Q 

v	$d[v]$	$\pi[v]$
A	∞	/
B	∞	/
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

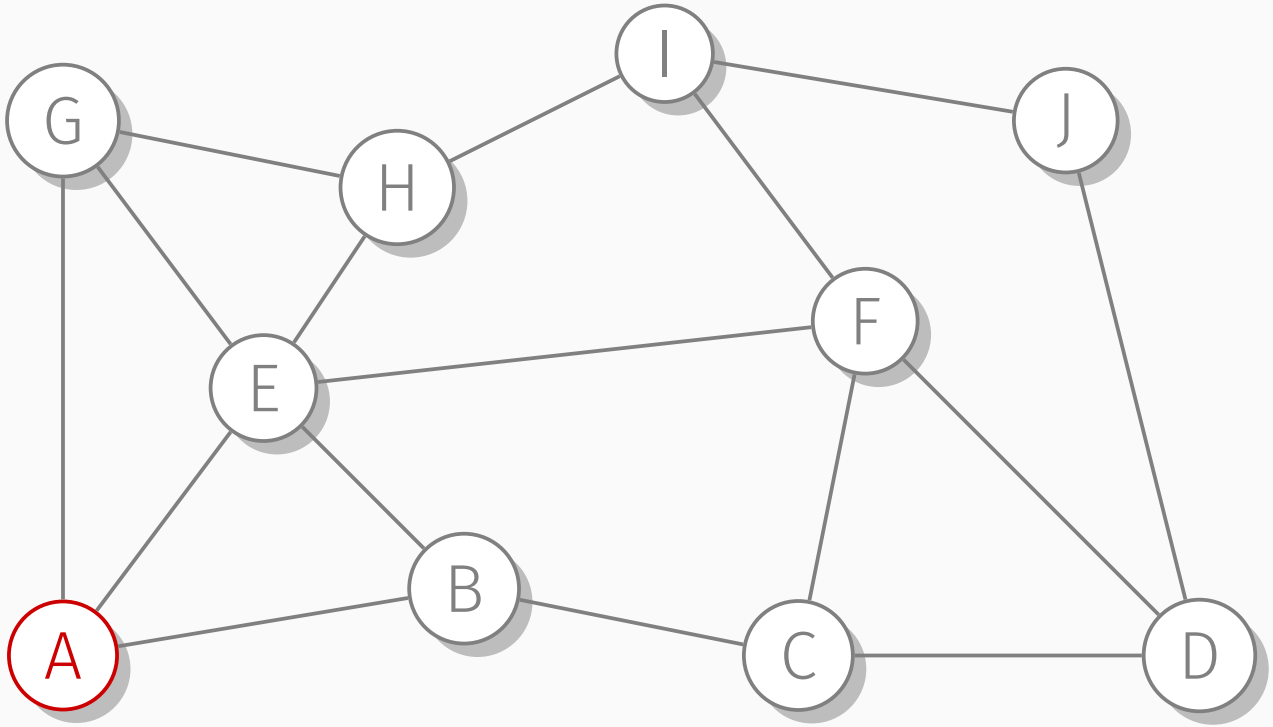
Průchod grafem do šířky, krok 2



Q | A |

v	$d[v]$	$\pi[v]$
A	0	/
B	∞	/
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

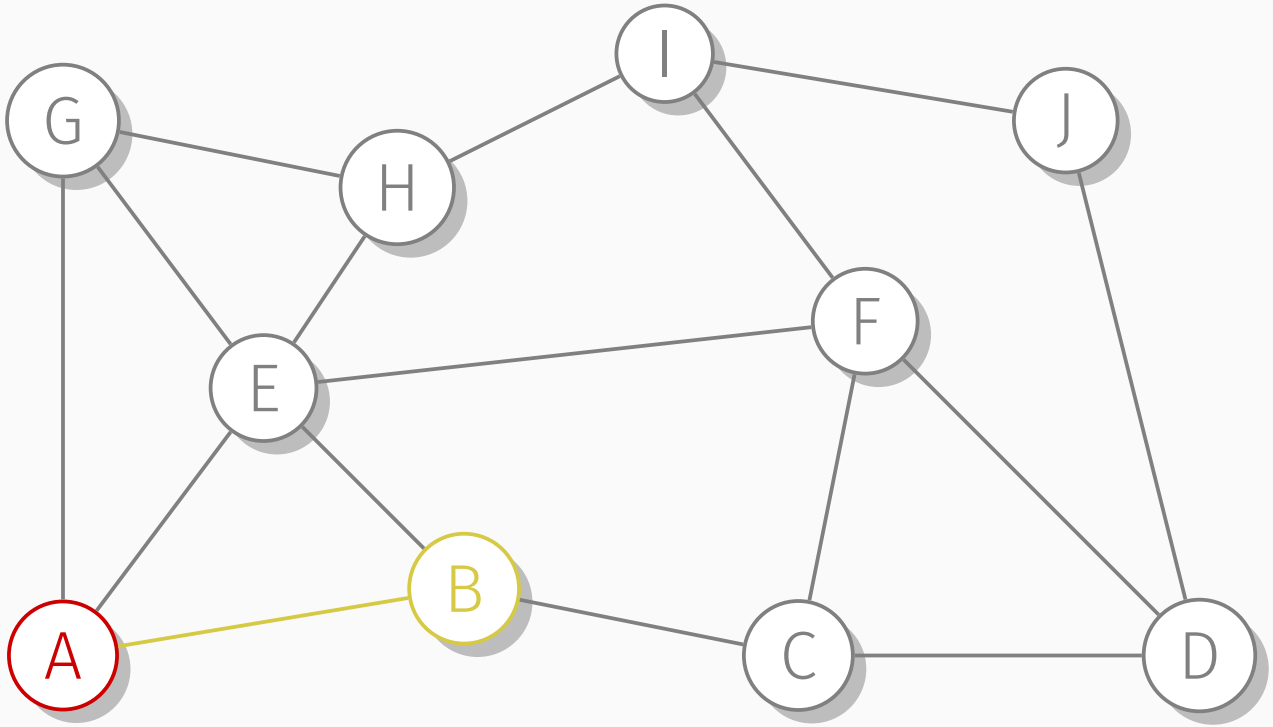
Průchod grafem do šířky, krok 3



Q 

v	$d[v]$	$\pi[v]$
A	0	/
B	∞	/
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

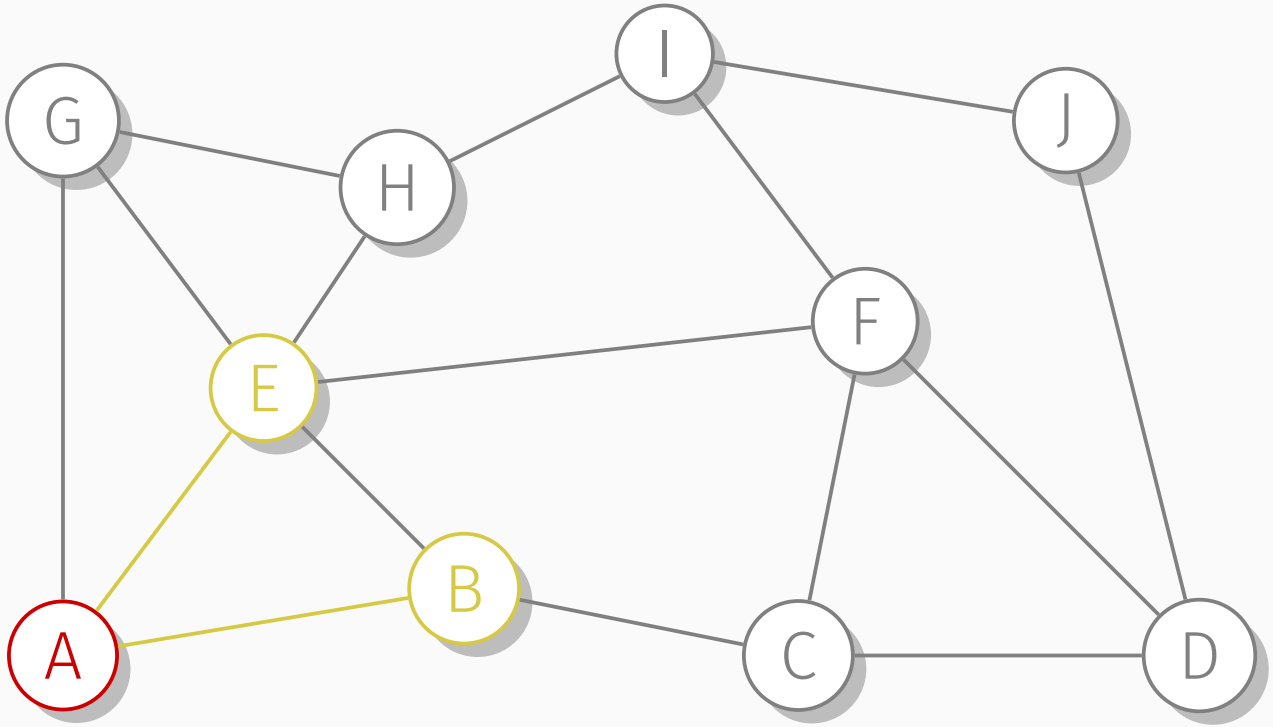
Průchod grafem do šířky, krok 4



Q | B |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

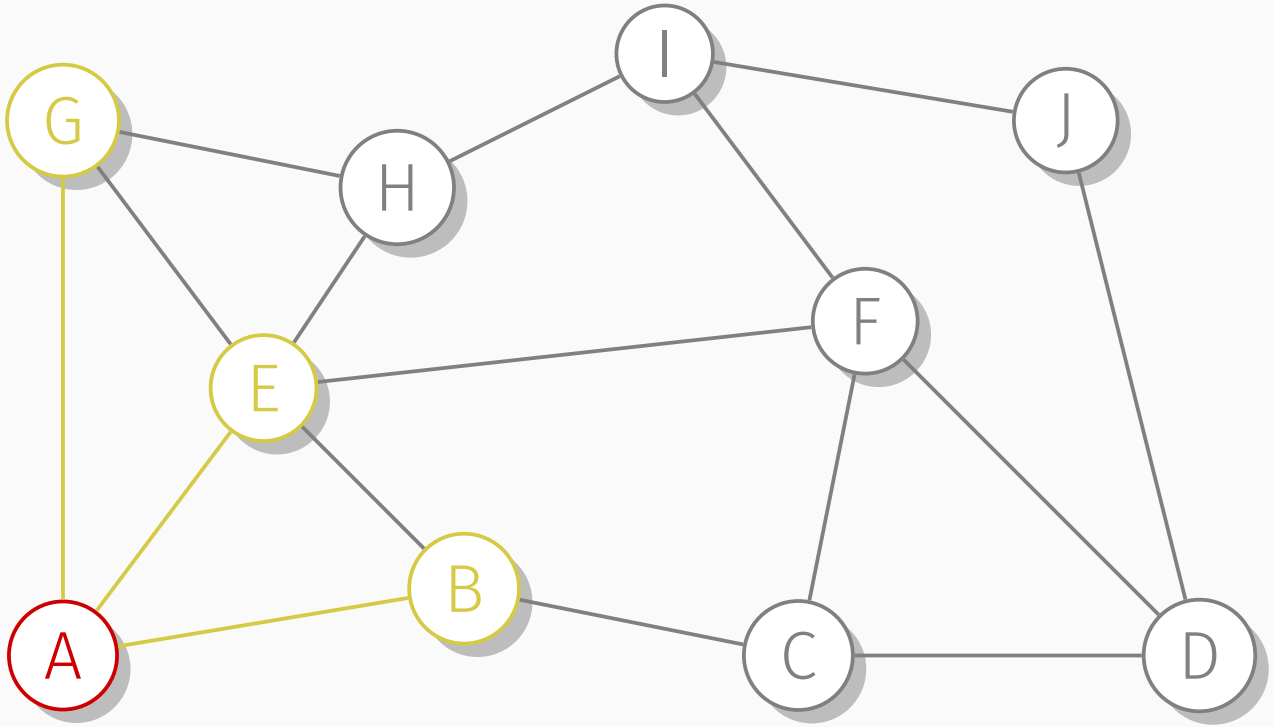
Průchod grafem do šířky, krok 5



Q | B | E |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

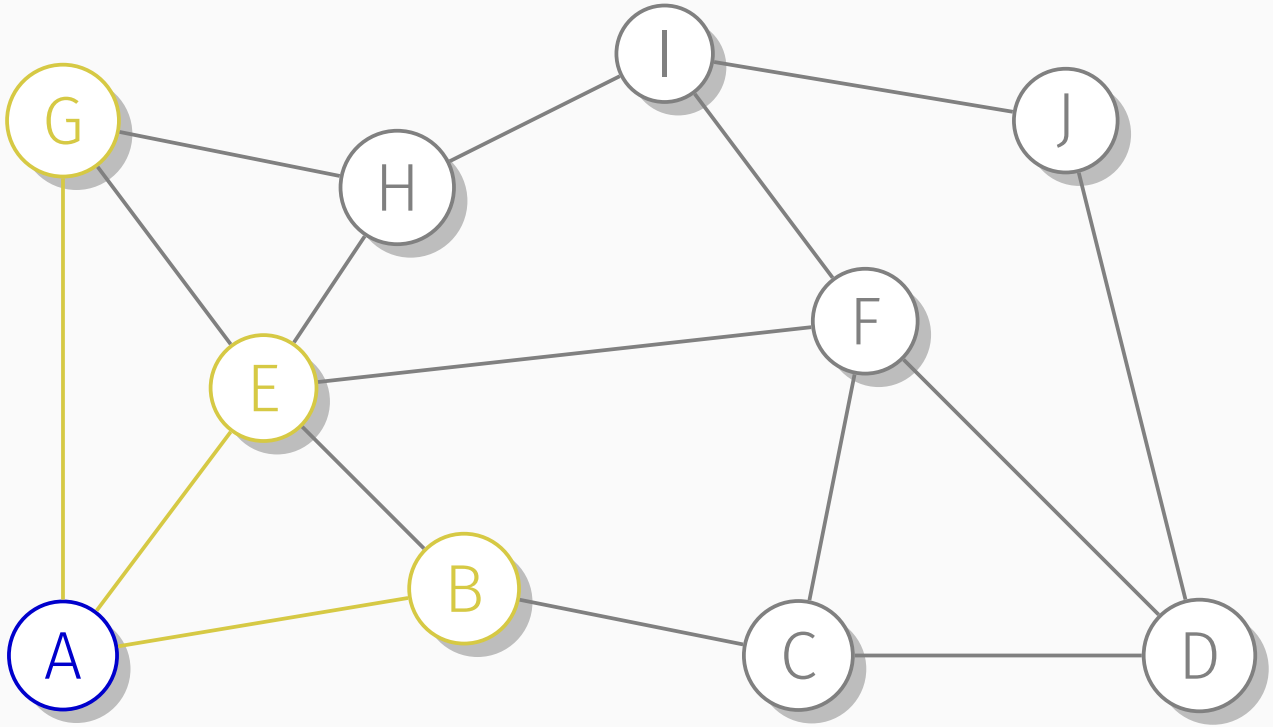
Průchod grafem do šířky, krok 6



Q | B | E | G |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

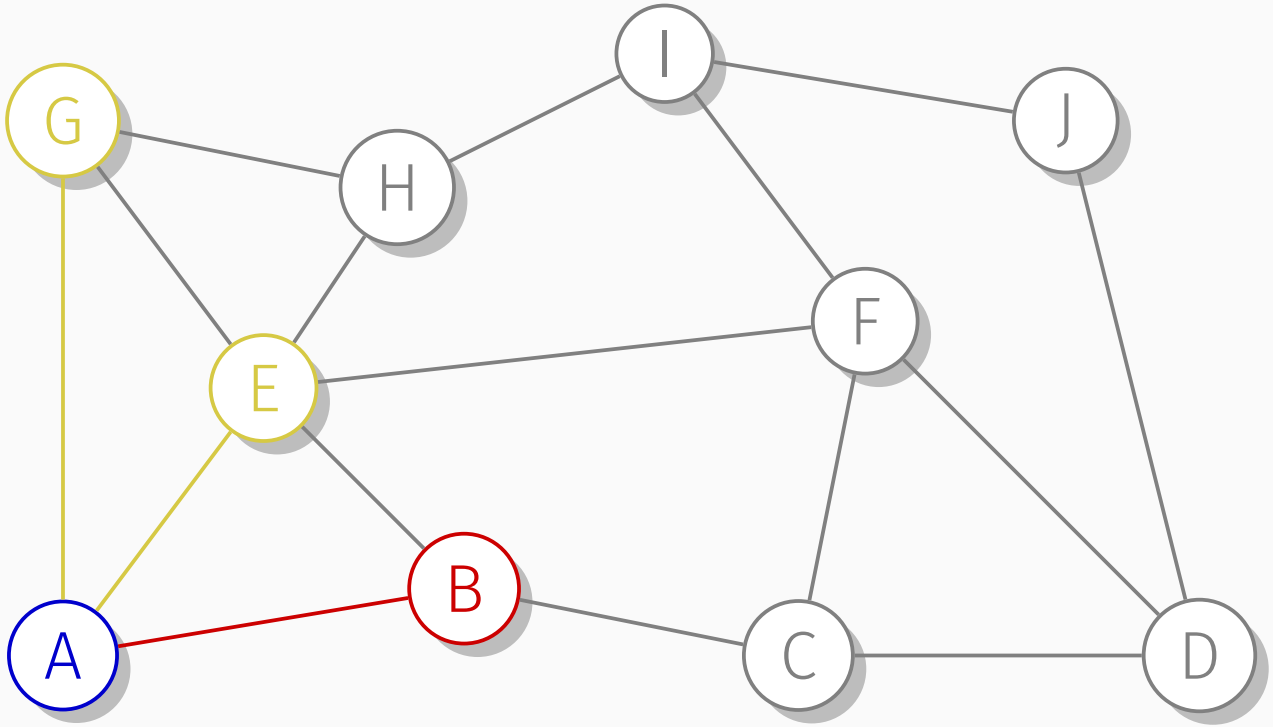
Průchod grafem do šířky, krok 7



Q | B | E | G |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

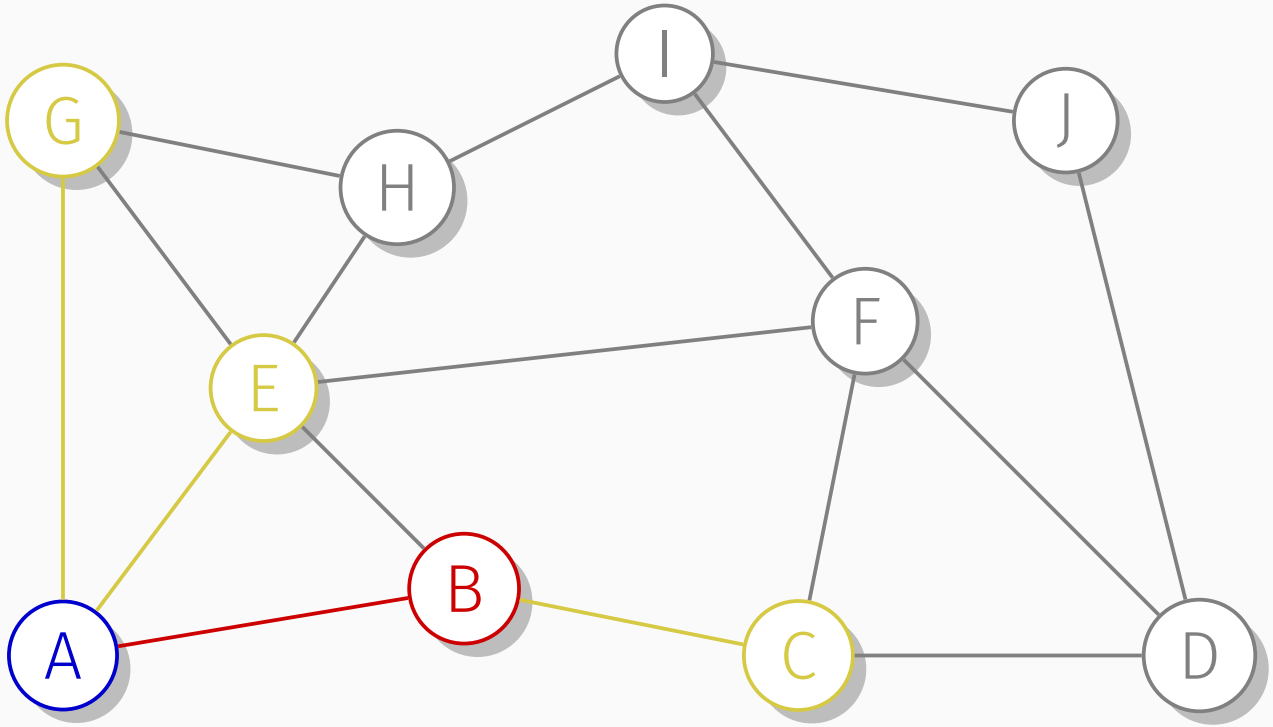
Průchod grafem do šířky, krok 8



Q | E | G |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

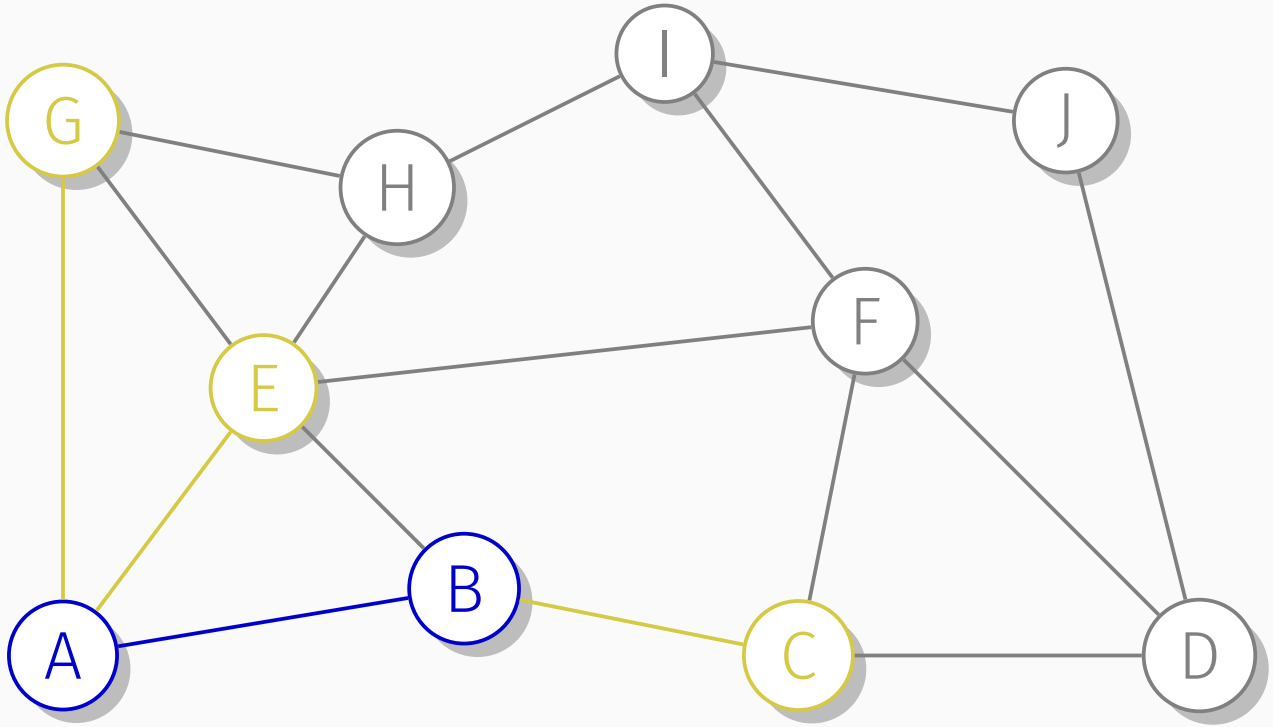
Průchod grafem do šířky, krok 9



Q | E | G | C |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

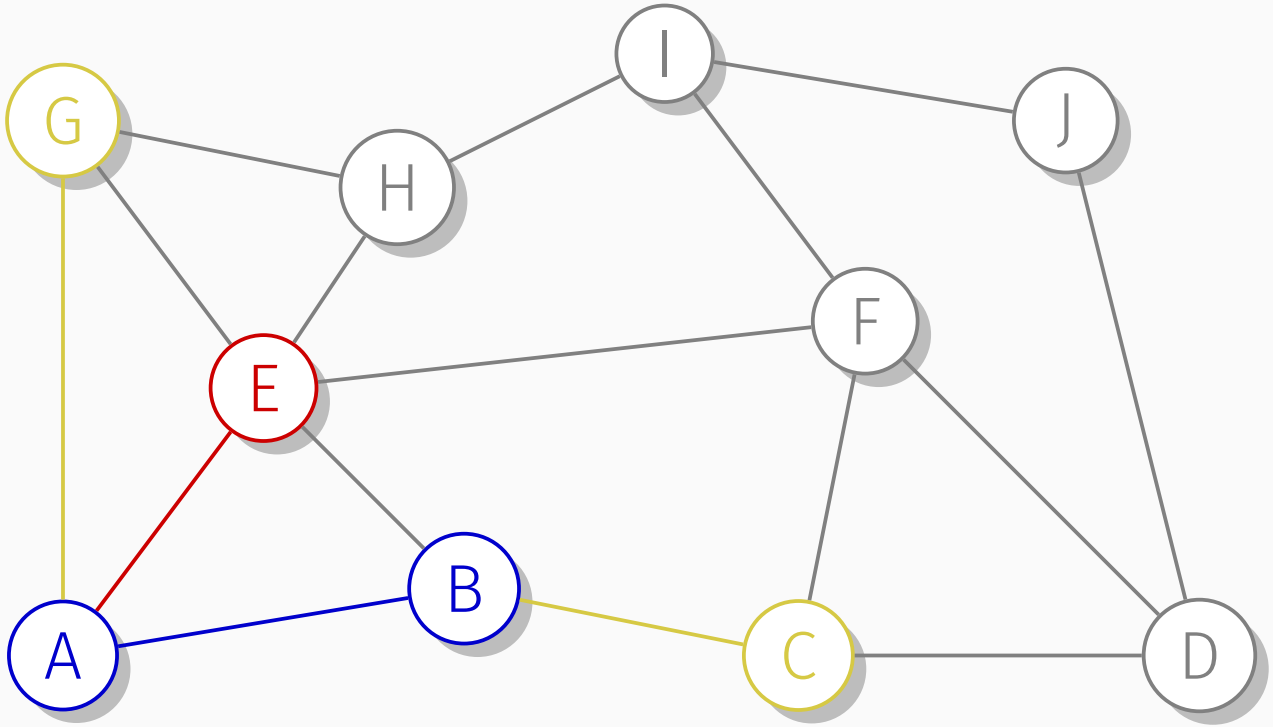
Průchod grafem do šířky, krok 10



Q | E | G | C |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

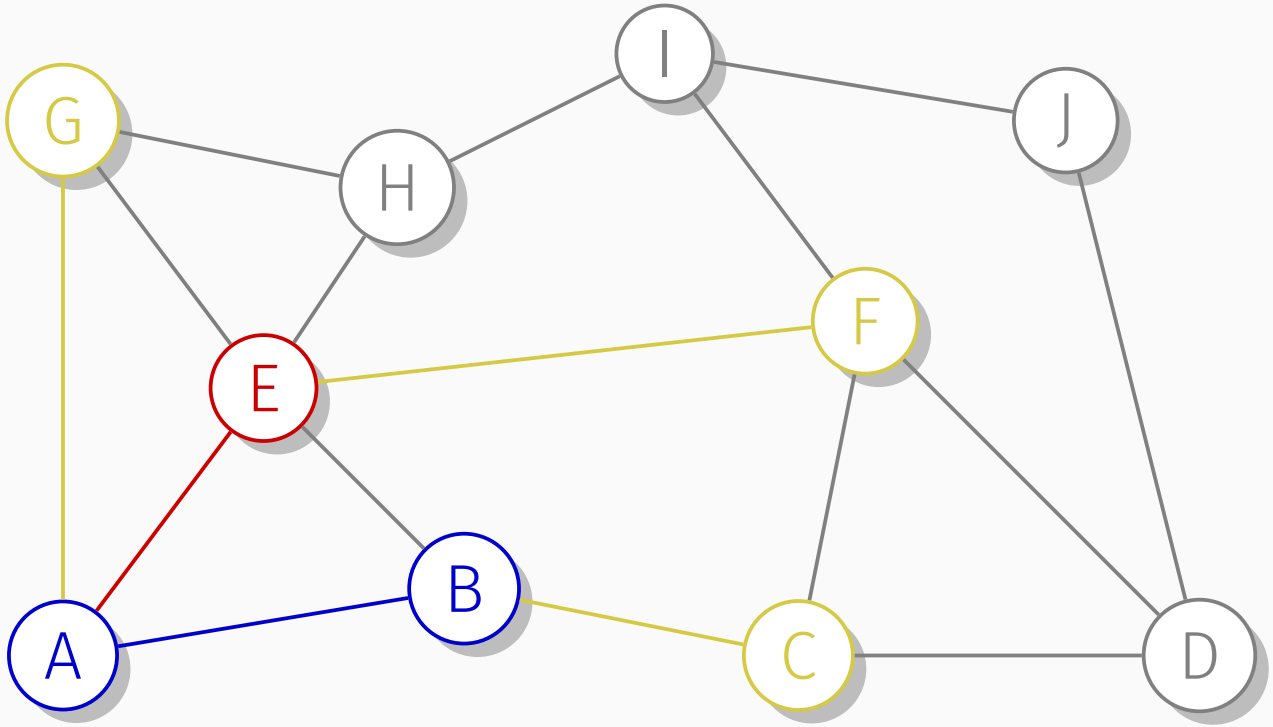
Průchod grafem do šířky, krok 11



Q | G | C |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

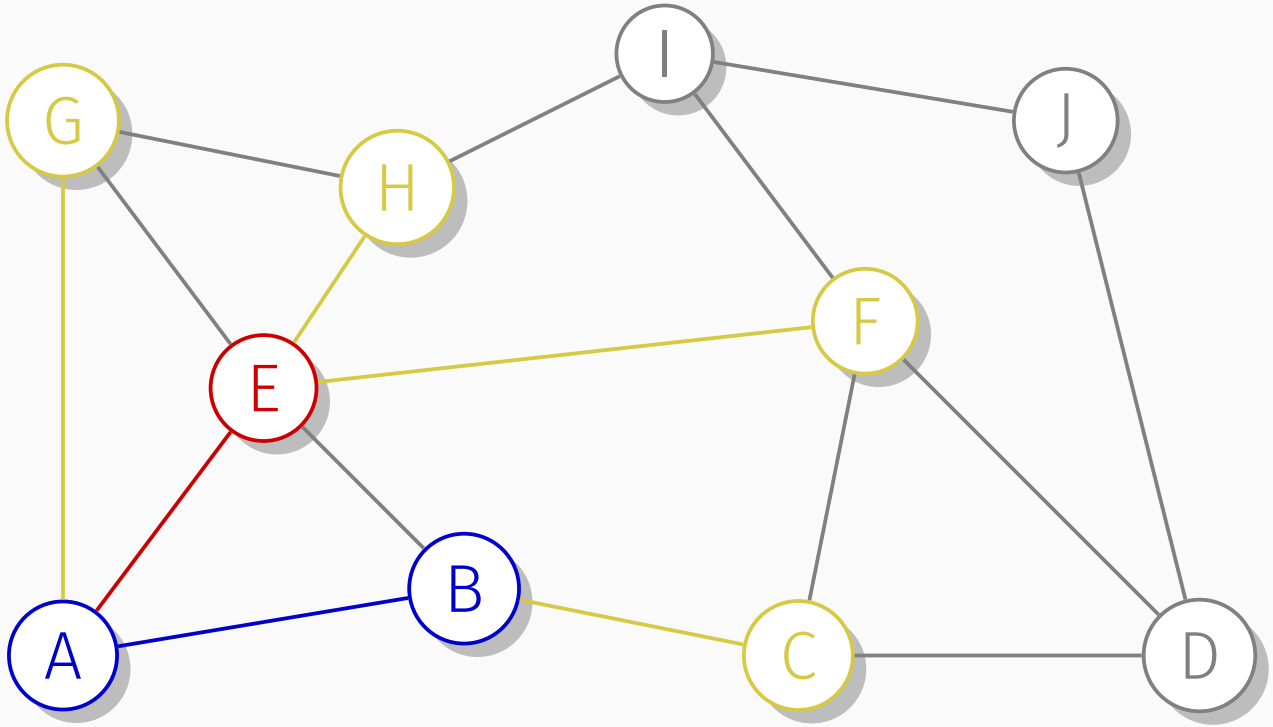
Průchod grafem do šířky, krok 12



Q | G | C | F |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	∞	/
I	∞	/
J	∞	/

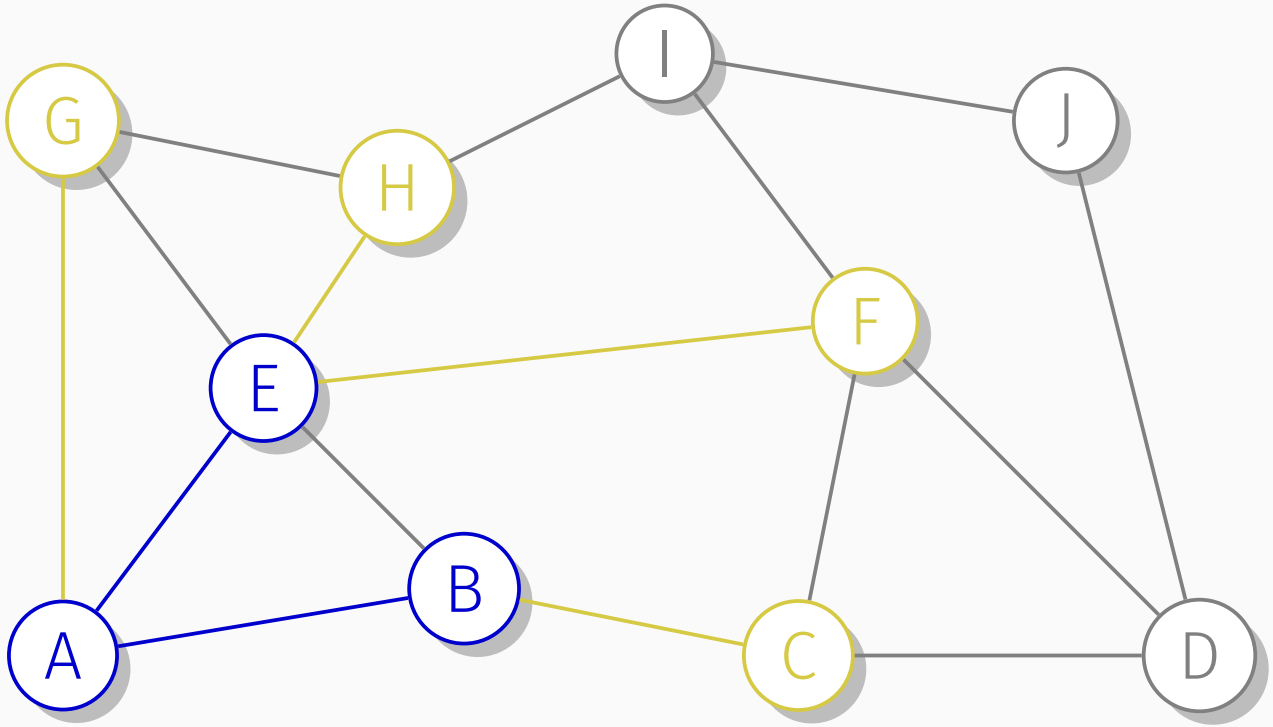
Průchod grafem do šířky, krok 13



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

Q | G | C | F | H |

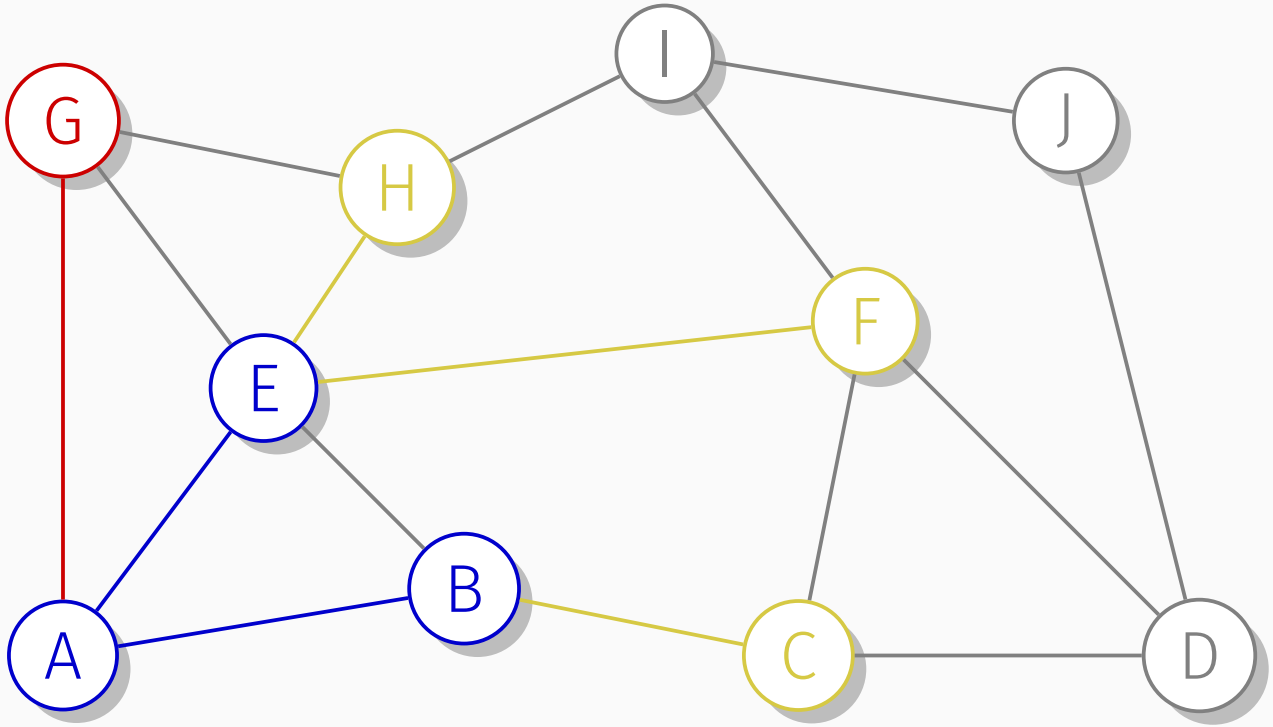
Průchod grafem do šířky, krok 14



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

Q | G | C | F | H |

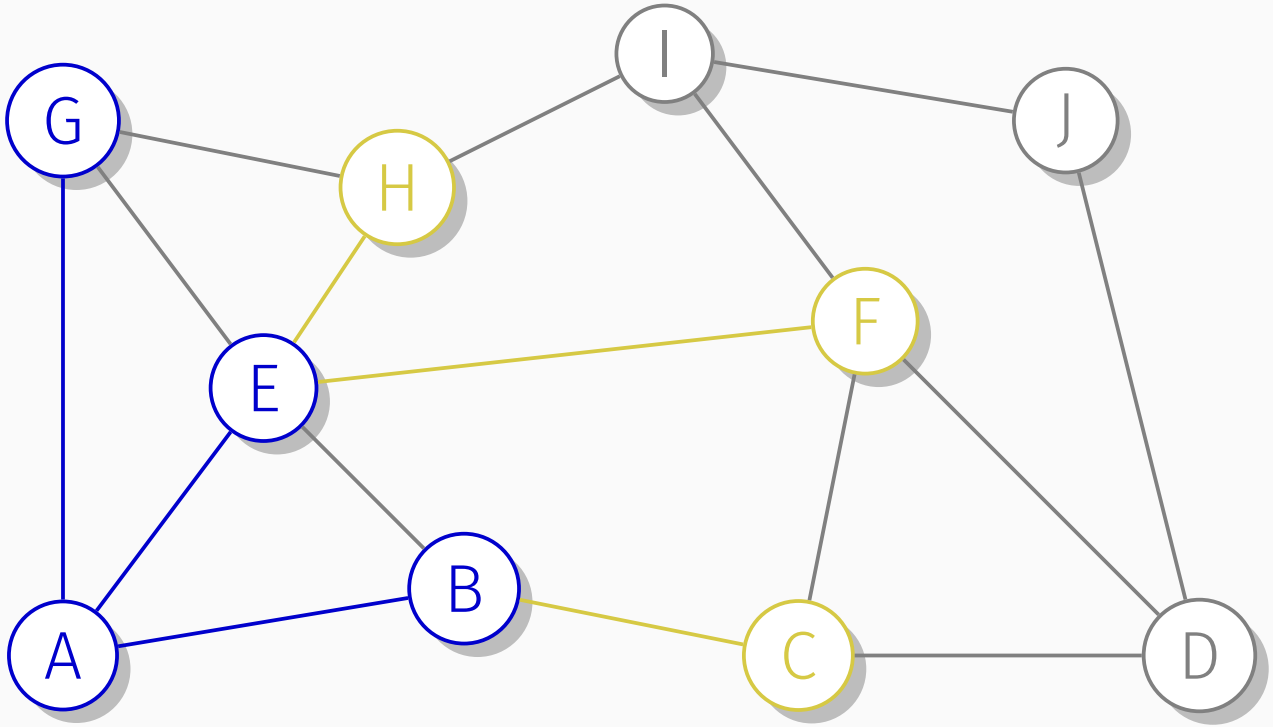
Průchod grafem do šířky, krok 15



Q | C | F | H |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

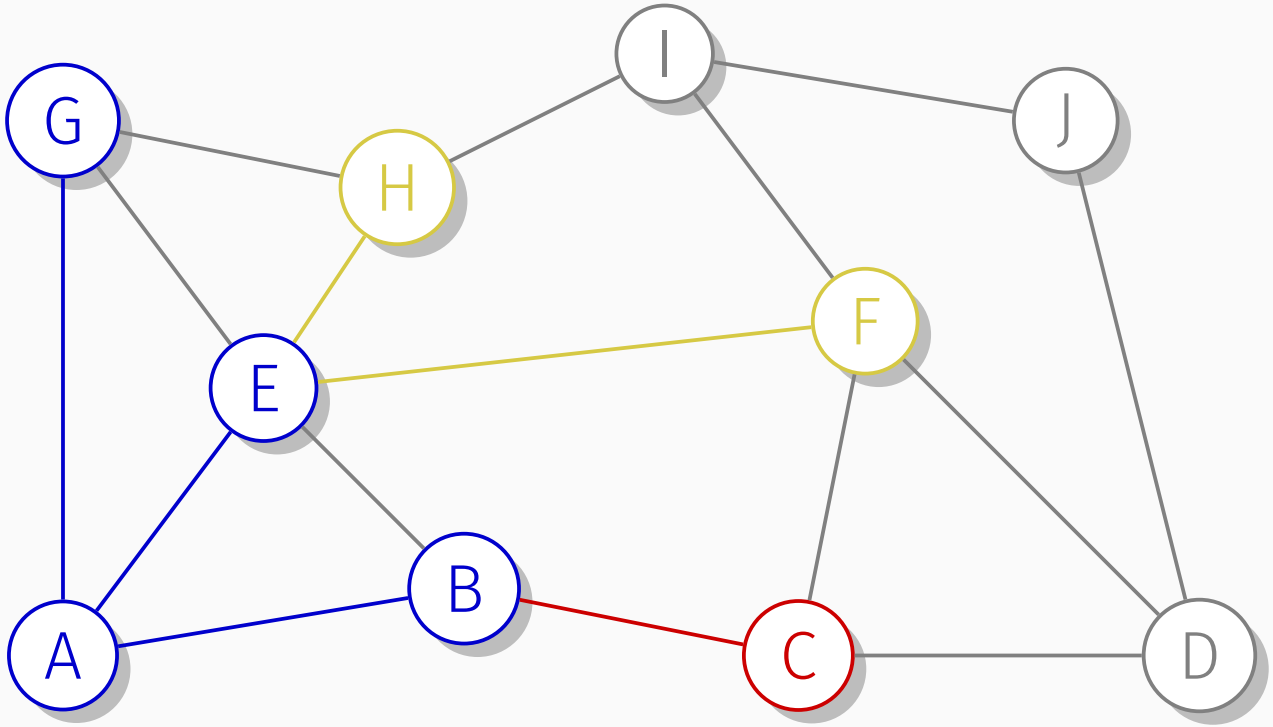
Průchod grafem do šířky, krok 16



Q | C | F | H |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

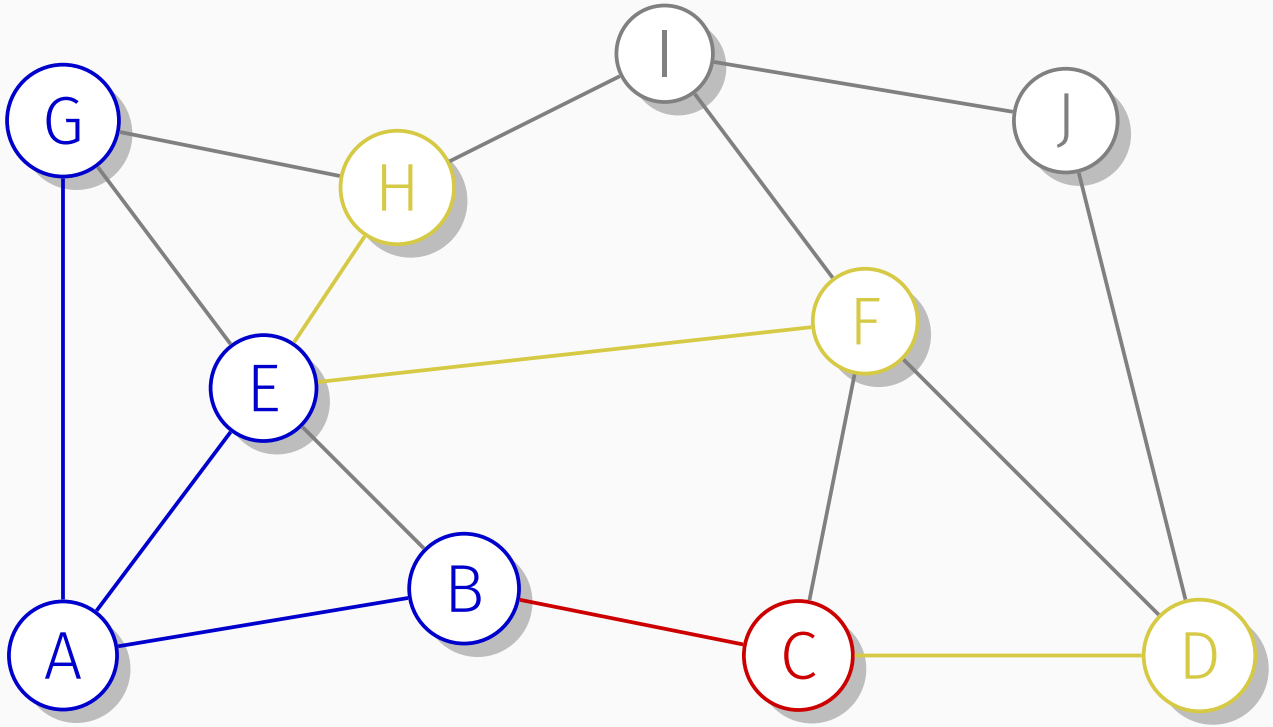
Průchod grafem do šířky, krok 17



Q | F | H |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

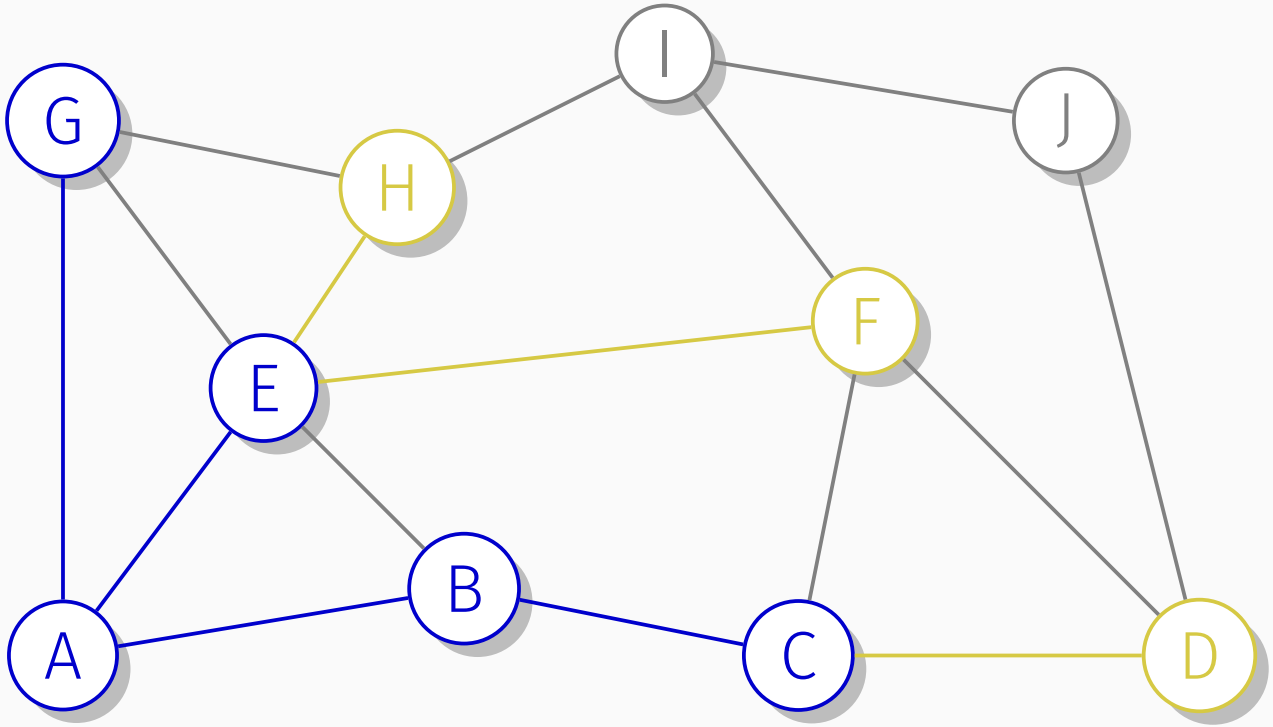
Průchod grafem do šířky, krok 18



Q | F | H | D |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

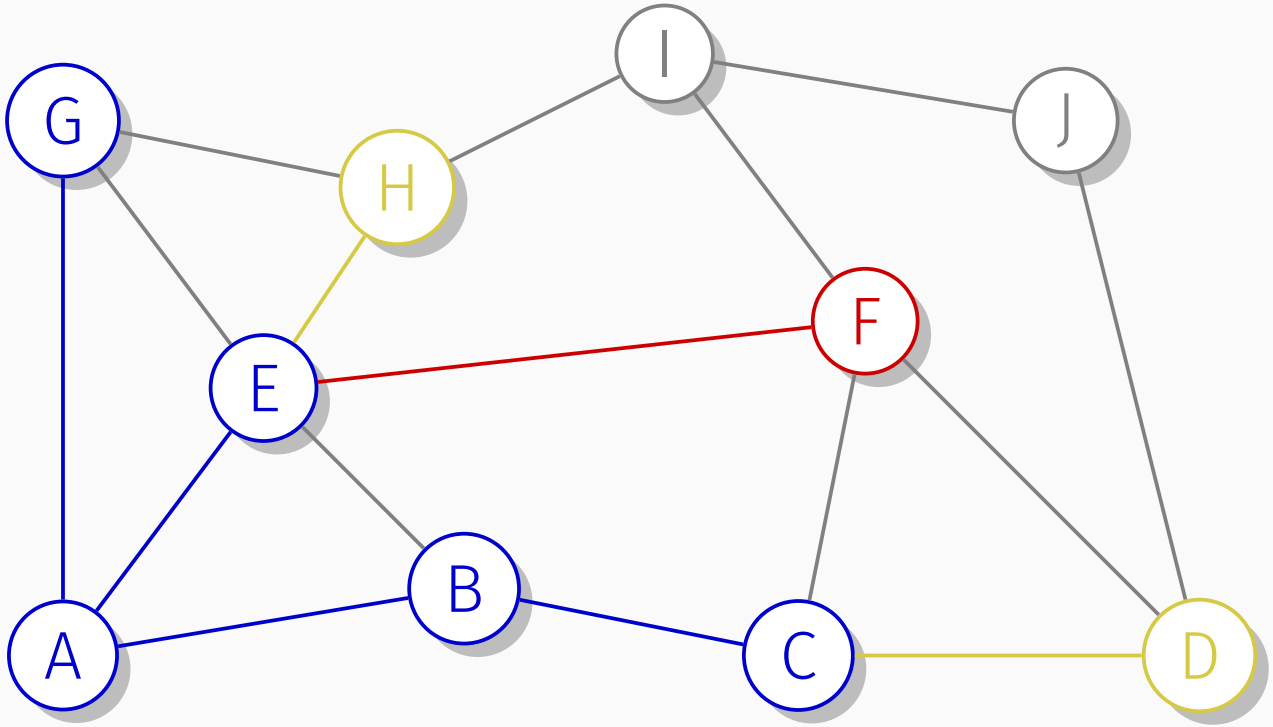
Průchod grafem do šířky, krok 19



Q | F | H | D |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

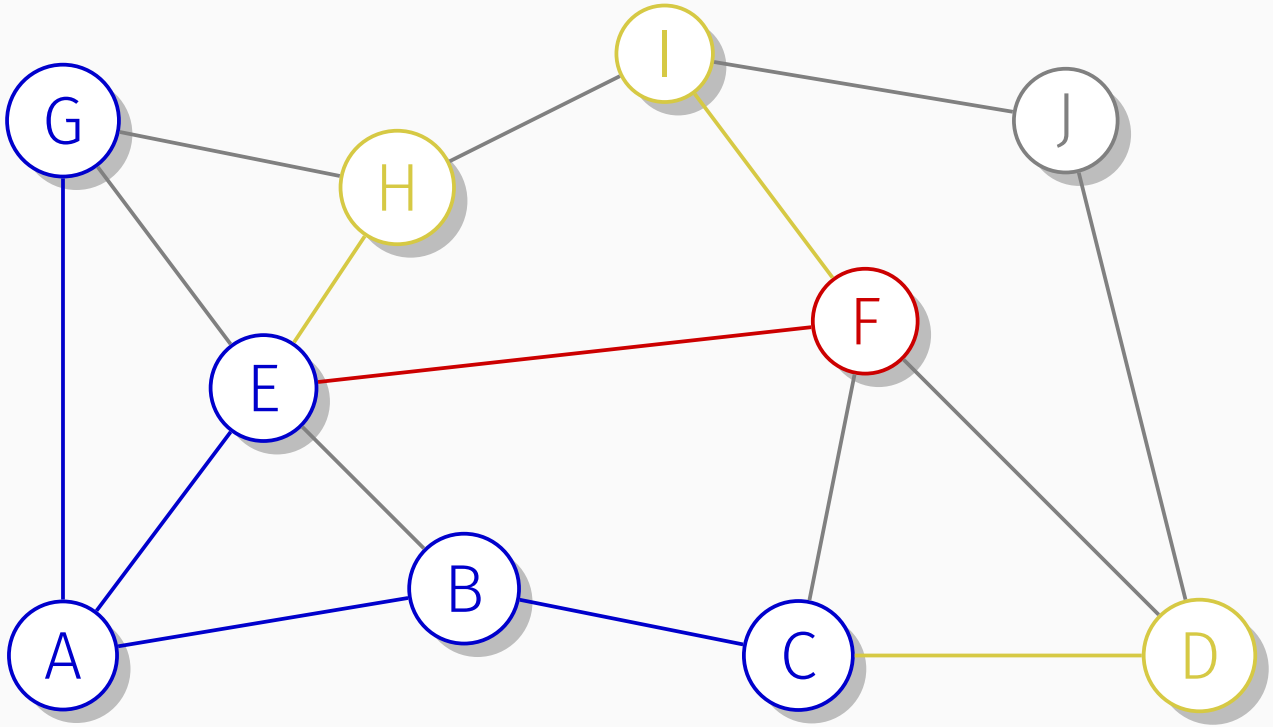
Průchod grafem do šířky, krok 20



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

Q | H | D |

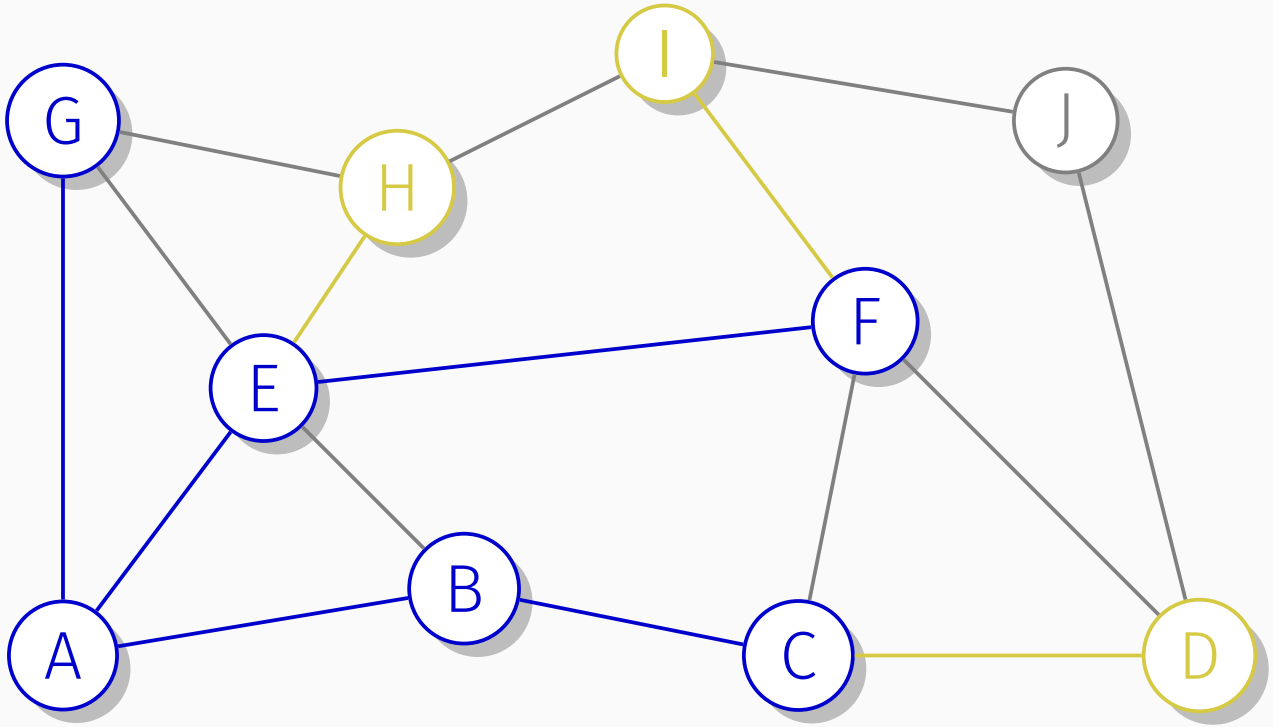
Průchod grafem do šířky, krok 21



Q | H | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

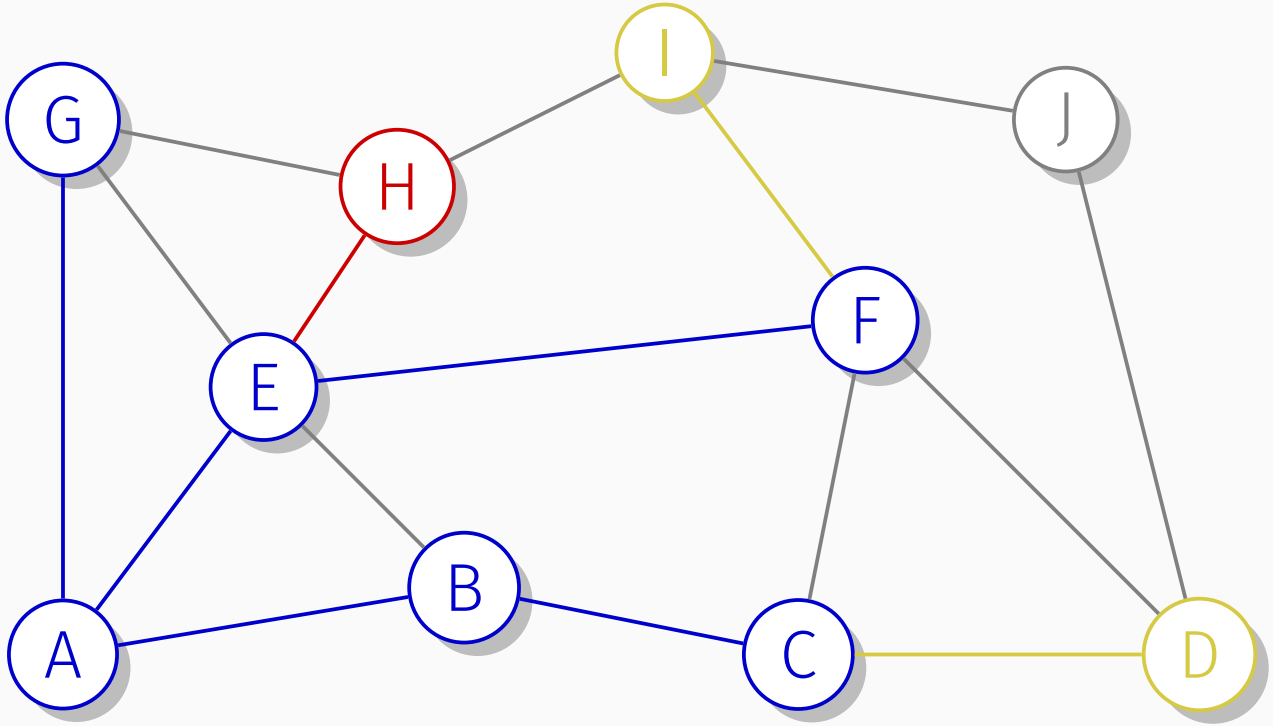
Průchod grafem do šířky, krok 22



Q | H | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

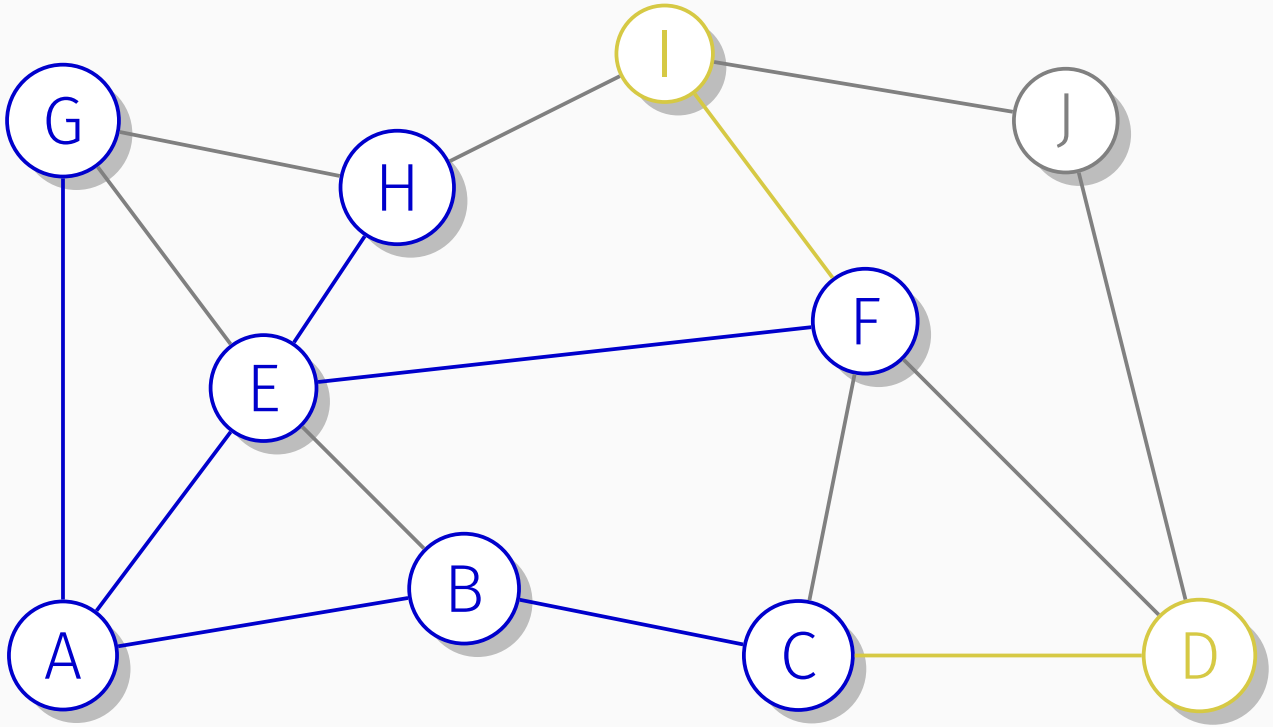
Průchod grafem do šířky, krok 23



Q | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

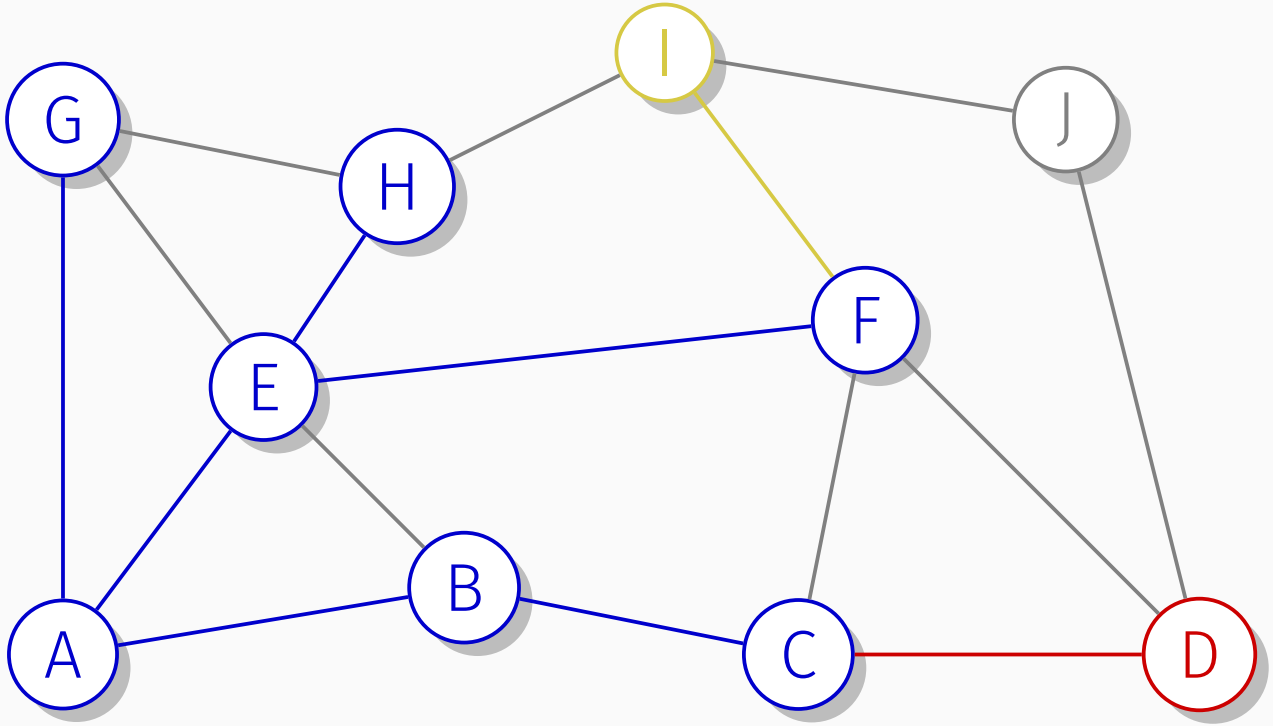
Průchod grafem do šířky, krok 24



Q | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

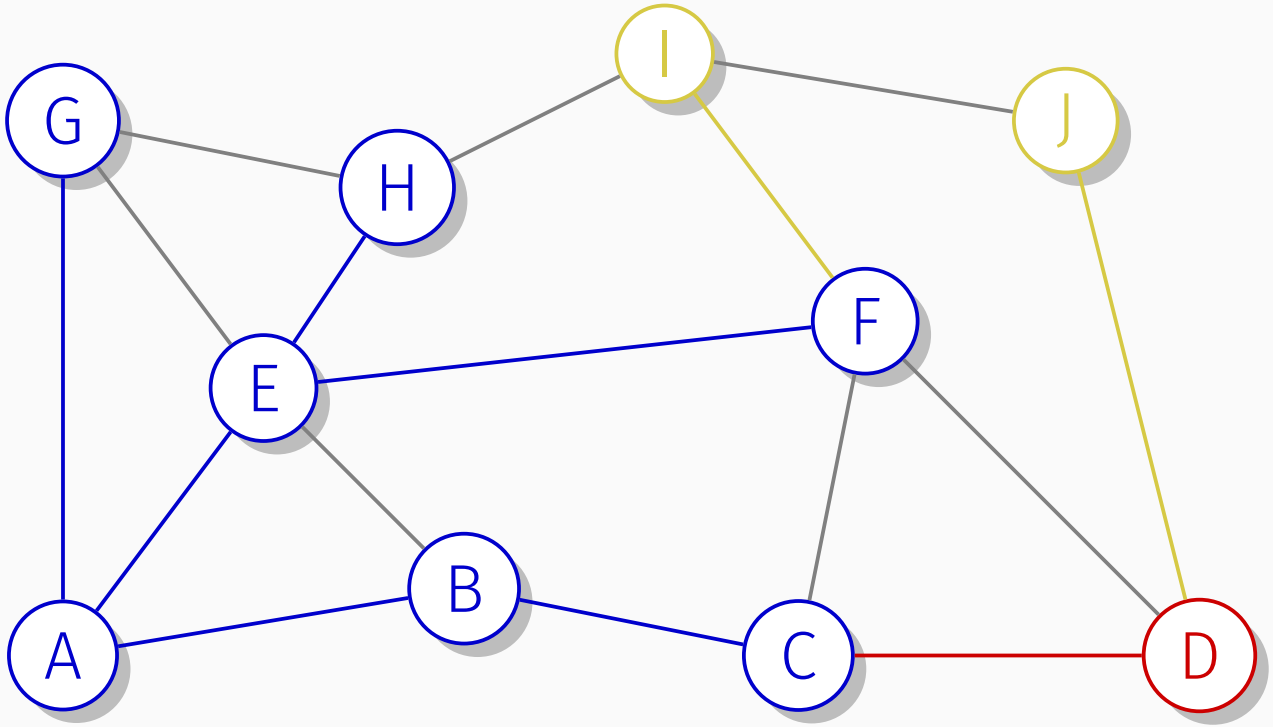
Průchod grafem do šířky, krok 25



Q | I |

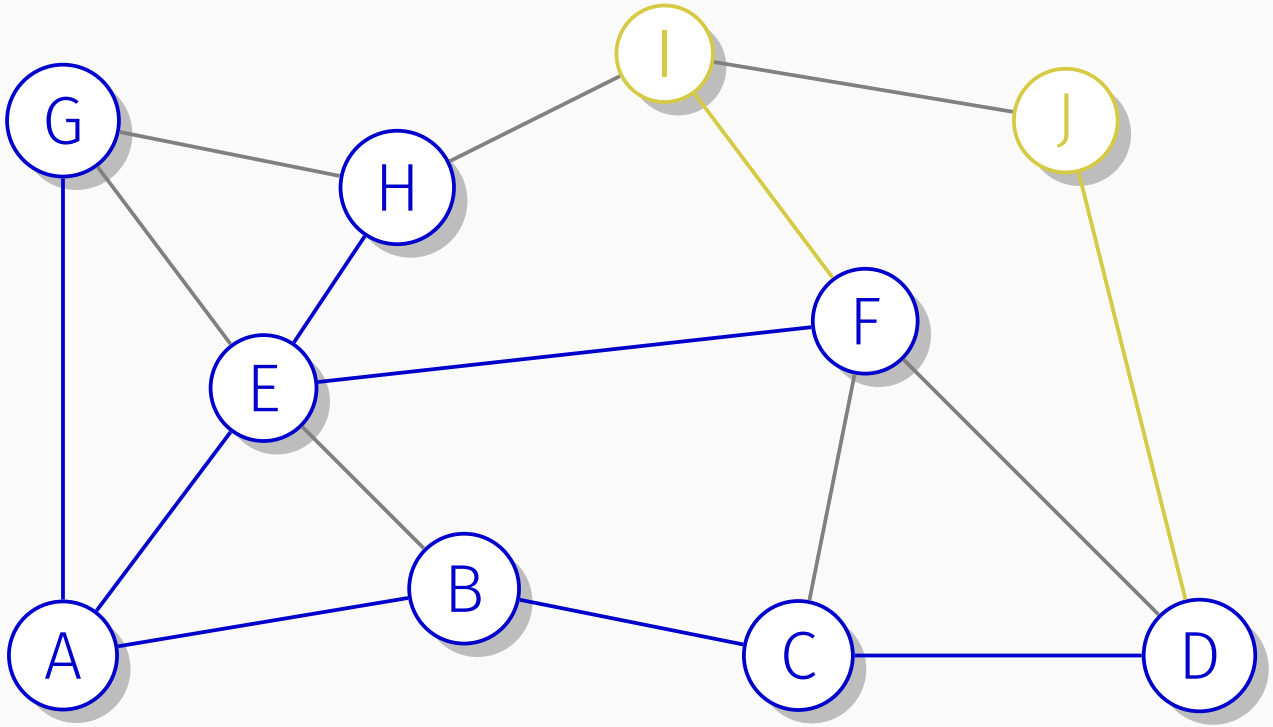
v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

Průchod grafem do šířky, krok 26



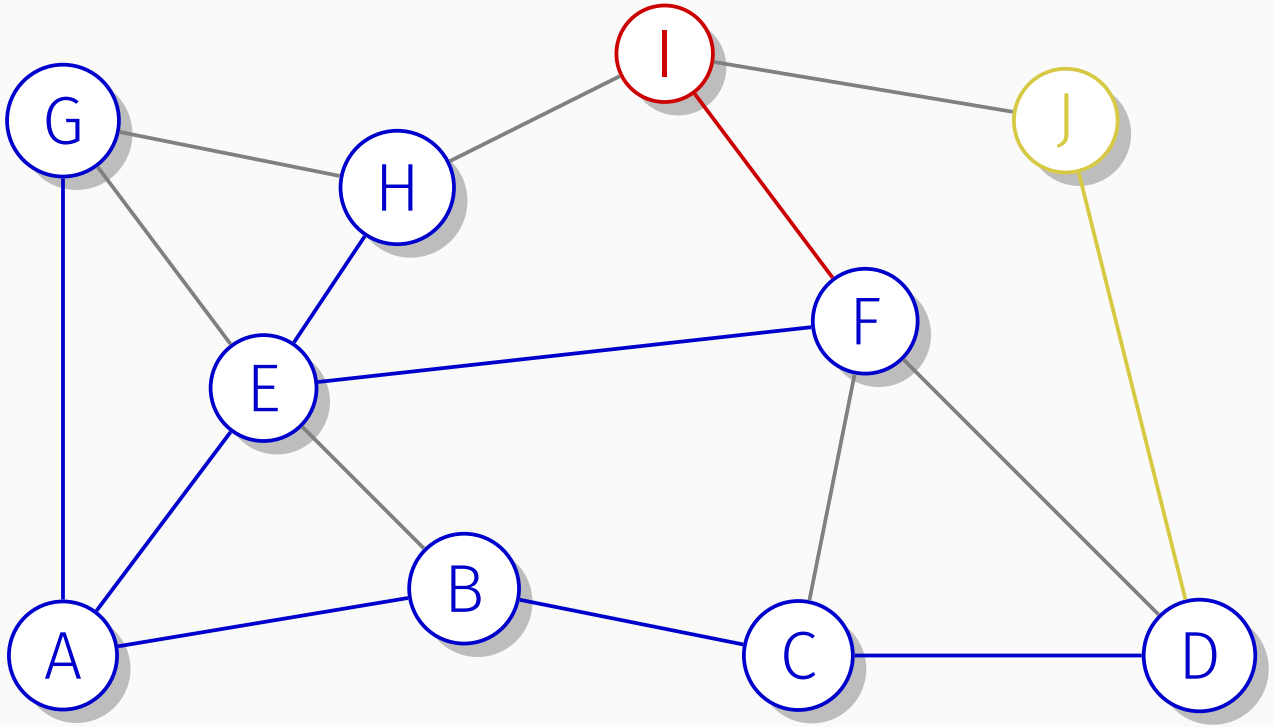
v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

Průchod grafem do šířky, krok 27



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

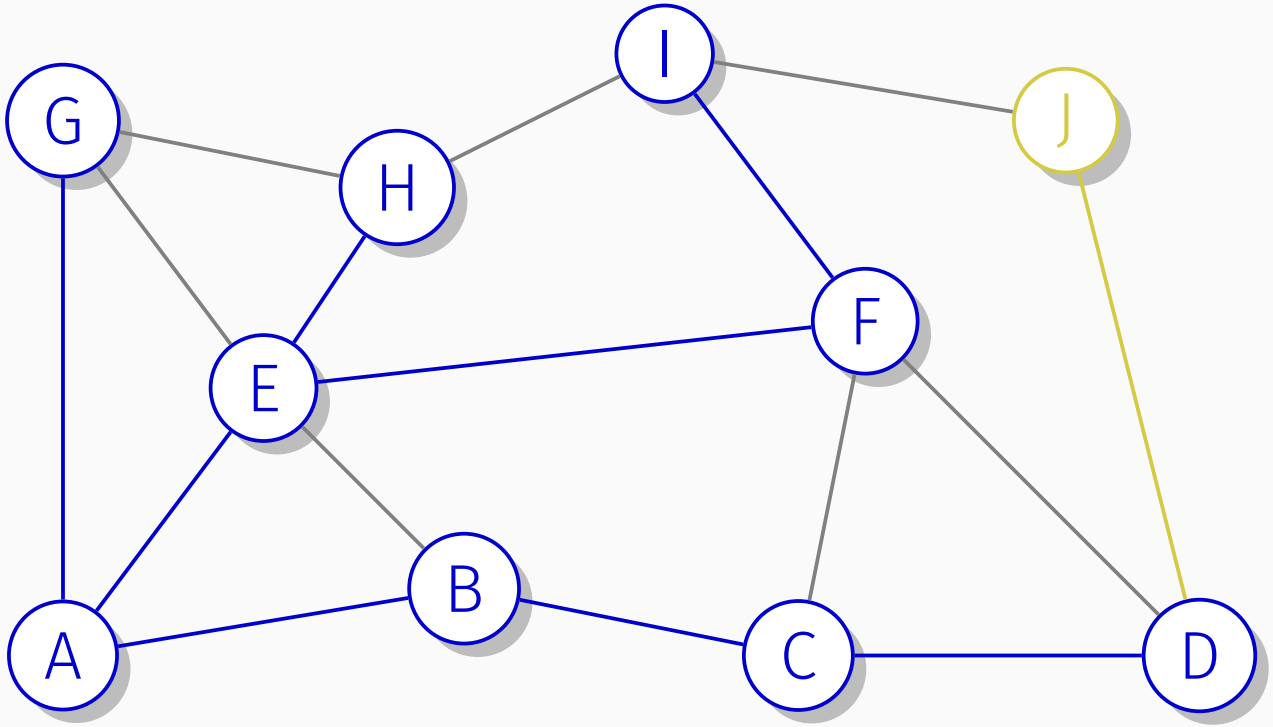
Průchod grafem do šířky, krok 28



Q | J |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

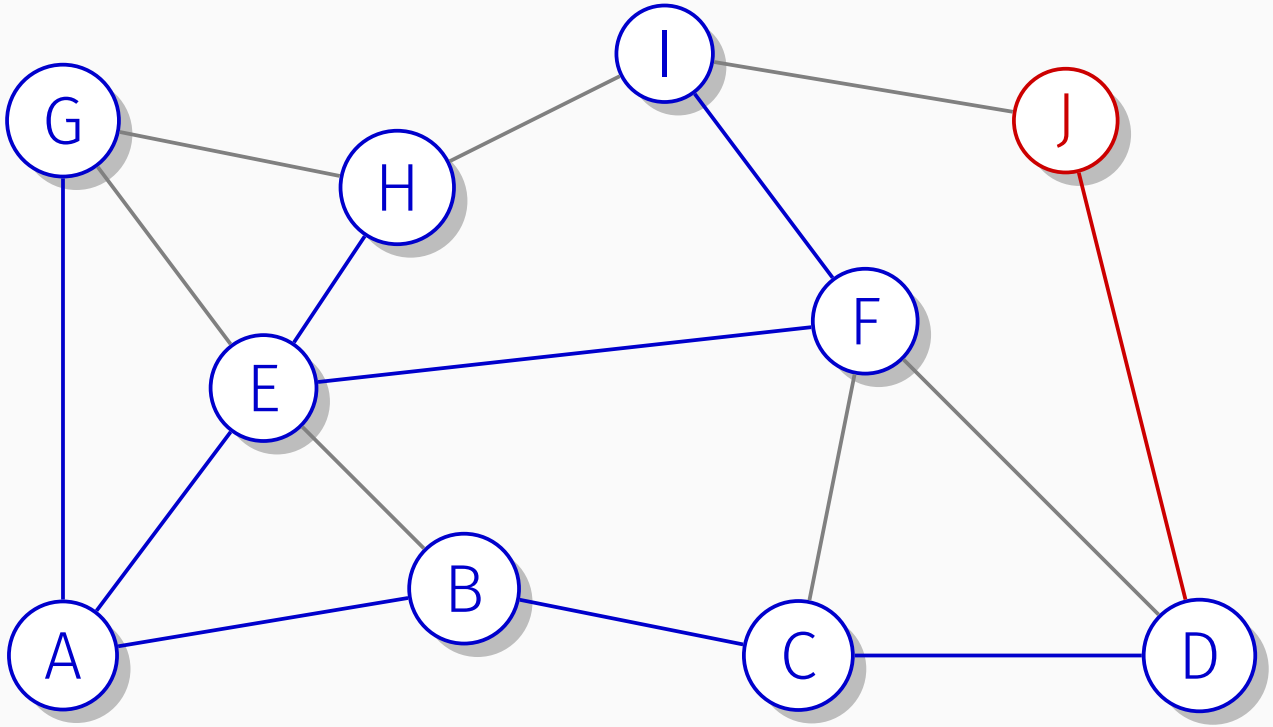
Průchod grafem do šířky, krok 29



Q | J |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

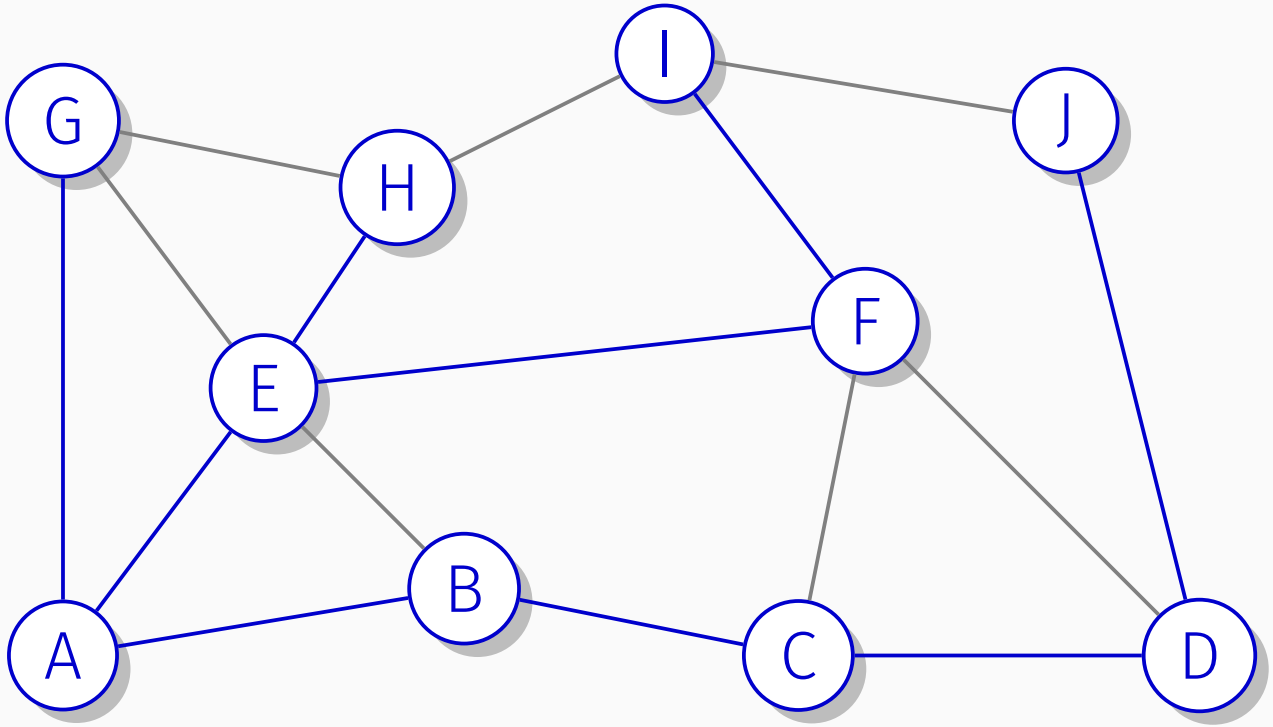
Průchod grafem do šířky, krok 30



Q

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

Průchod grafem do šířky, krok 31



Q |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

Průchod grafem do hloubky a do šířky

Animace

K oběma algoritmům průchodu grafem jsou dostupné dvě animace:

- vyhledání cesty v bludišti a
- aplikace v počítačové grafice – vyplňování oblasti.

Animace jsou k dispozici animace v samostatných souborech.

Zdroje pro samostatné studium

- Strategie řešení problémů hrubou silou
  - kniha [2], kapitola 3, strany 97 – 98
- Třídění výběrem
  - kniha [2], kapitola 3.1, strany 98 – 100
- Bublinové třídění
  - kniha [2], kapitola 3.1, strany 100 – 101
- Třídění přetřásáním
  - kniha [4], kapitola 4, strany 78 – 79
- Sekvenční vyhledávání
  - kniha [2], kapitola 3.2, strany 104 – 104
- Vyhledávání podřetězce hrubou silou
  - kniha [2], kapitola 3.2, strany 105 – 106

Zdroje pro samostatné studium (pokrač.)

- Problém nejbližší dvojice bodů
  - kniha [2], kapitola 3.3, strany 108 – 109
- Konvexní obal množiny
  - kniha [2], kapitola 3.3, strany 109 – 113
- Úplné prohledávání
  - kniha [2], kapitola 3.4, strany 115
- Problém obchodního cestujícího
  - kniha [2], kapitola 3.4, strany 116
- Problém batohu
  - kniha [2], kapitola 3.4, strany 116 – 117
- Průchod grafem do hloubky
  - kniha [2], kapitola 3.5, strany 122 – 125
- Průchod grafem do šířky
  - kniha [2], kapitola 3.5, strany 125 – 128

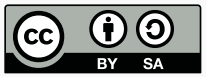
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Strategie řešení sniž a vyřeš

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Strategie řešení sniž a vyřeš

Třídění vkládáním – InsertSort

Strategie řešení sniž a vyřeš

Topologické třídění

Strategie řešení sniž a vyřeš

Generování kombinatorických objektů

Generování kombinatorických objektů

- Generování kombinací, variací, permutací, podmnožin je součástí různých algoritmů.
- Typicky jde o výběr nějaké alternativy, možnosti, nastavení parametrů...
- Příklady – Problém obchodního cestujícího, Problém batohu.
- Matematika se zajímá spíše počty těchto objektů, zatímco informatika hledá algoritmy jak je generovat.
- Počet těchto objektů roste exponenciálně nebo i rychleji!

Generování permutací

- Budeme generovat permutace celých čísel $1, 2, \ldots, n$.
- Obecněji můžeme generovat permutací prvků $\{a_1, a_2, \ldots, a_n\}$.
- Využití strategie sniž a vyřeš:
  1. Generování $n!$ permutací pro n prvků převedeme na generování $(n-1)!$ permutací $n-1$ prvků.
  2. Jakmile máme problém pro $n-1$ vyřešen, vložíme prvek n na všech n možných pozic do každé z $(n-1)!$ permutací.
  3. Jinak řečeno, máme $(n-1)!$ permutací, ke každé z nich vygenerujeme n dalších. Celkově dostáváme $n(n-1)! = n!$ permutací.

Generování permutací – příklad

permutace prvku 1	1		
vložení 2 do permutace 1 zprava doleva	1**2**	**2**1	
vložení 3 do permutace 12 zprava doleva	12**3**	1**3**2	**3**12
vložení 3 do permutace 21 zleva doprava	**3**21	2**3**1	21**3**

Co je z příkladu patrné?

- Všechny permutace jsou navzájem různé.
- Minimální změna mezi permutacemi – dvě po sobě jdoucí permutace se liší výměnou jediné dvojice prvků a to dokonce sousedních prvků.

Johnson-Trotterův algoritmus

- Existuje možnost generovat permutace n prvků? Bez nutnosti vygenerovat permutace pro $n - 1$? Ano, existuje.
- Každému z n prvků permutace přiřadíme **šipku** (směr), buď vlevo nebo vpravo.
- O prvku k řekneme, že je v dané permutaci **mobilní**, jestliže sousední prvek ve směru šipky prvku k je menší než k.

Příklad

Permutace se šipkami

$$\vec{3}\ \overleftarrow{2}\ \vec{4}\ \vec{1}$$

Prvky 3 a **4 jsou mobilní**, prvky 2 a **1 nejsou mobilní**.

Johnson-Trotterův algoritmus

Vstup : Přirozené číslo n
Výstup: Seznam všech permutací čísel $\{1, \ldots, n\}$

1 $\pi \leftarrow \overleftarrow{1}\ \overleftarrow{2}\ \ldots\ \overleftarrow{n}$;
2 **while** π obsahuje mobilní prvek **do**
3 $\quad k \leftarrow$ největší mobilní prvek v π;
4 $\quad$ přehoď v π prvek k se sousedem ve směru šipky;
5 $\quad$ změň směr šipky u všech prvků větších než k;
6 $\quad$ vlož nově vytvořenou permutaci (krok 4) π do výsledného seznamu;
7 **end**
8 **return** seznam všech permutací;

Johnson-Trotterův algoritmus – příklad

Ukázka generování permutací pro $n = 3$

$$
\begin{array}{ccc}
\overleftarrow{1} & \overleftarrow{2} & \overleftarrow{3} \\
\overleftarrow{1} & \overleftarrow{3} & \overleftarrow{2} \\
\overleftarrow{3} & \overleftarrow{1} & \overleftarrow{2} \\
\overrightarrow{3} & \overleftarrow{2} & \overleftarrow{1} \\
\overleftarrow{2} & \overrightarrow{3} & \overleftarrow{1} \\
\overleftarrow{2} & \overleftarrow{1} & \overrightarrow{3}
\end{array}
$$

O prvku k řekneme, že je v dané permutaci **mobilní**, jestliže sousední prvek ve směru šipky prvku k je menší než k.

Johnson-Trotterův algoritmus – příklad, $n = 4$

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

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

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

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

Johnson-Trotterův algoritmus

- Jeden z nejefektivnějších algoritmů pro generování permutací.
- Časová složitost algoritmu je $\Theta(n!)$.
- „Děsivá“ složitost algoritmu ale není způsobena algoritmem samotným, ten pracuje velice rychle. Je zdpůsobena obrovským množstvím permutací, které musí vygenerovat...

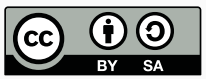
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Strategie řešení rozděl a panuj

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Strategie řešení rozděl a panuj

Master theorem

Strategie řešení rozděl a panuj

Násobení velkých celých čísel

Násobení velkých celých čísel

- Násobení „běžných“ celých čísel řeší procesor.
- Co násobení mnohem větších čísel, se stovkami číslic? Například v kryptografii.
- Určitě by šlo implementovat algoritmus podobný písemnému násobení.
- Jeho provedení vyžaduje n^2 násobení číslic, kde n je počet číslic.

		2	3
		1	4
—	—	—	—
		9	2
	2	3	0
—	—	—	—
	3	2	2

Otázka k řešení

Lze to provést rychleji? Nebo je toto nejlepší možný algoritmus?

Násobení velkých celých čísel – násobení 23 a 14

Určíme dekadický rozvoj čísel

$$\begin{aligned} 23 &= 2 \cdot 10^1 + 3 \cdot 10^0 \\ 14 &= 1 \cdot 10^1 + 4 \cdot 10^0 \end{aligned}$$

A oba rozvoje mezi sebou vynásobíme

$$\begin{aligned} 23 \times 14 &= \left(2 \cdot 10^1 + 3 \cdot 10^0\right) \times \left(1 \cdot 10^1 + 4 \cdot 10^0\right) \\ &= (2 \times 1) \cdot 10^2 + (2 \times 4 + 3 \times 1) \cdot 10^1 + (3 \times 4) \cdot 10^0 \\ &= 322 \end{aligned}$$

Pro výpočet jsme potřebovali 4 násobení (označena $\times$), tj. n^2 násobení.

Násobení velkých celých čísel – násobení 23 a 14 (pokrač.)

Prostřední výraz (desítky) lze vyhodnotit i takto

$$2 \times 4 + 3 \times 1 = (2 + 3) \times (1 + 4) - 2 \times 1 - 3 \times 4$$

Neviděli jsme výrazy 2×1 a 3×4 už někde...?

Obecněji mějme $a = a_1 a_0$ a $b = b_1 b_0$ potom

$$c = a \times b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0,$$

kde

- $c_2 = a_1 \times b_1$ je násobek prvních číslic,
- $c_0 = a_0 \times b_0$ je násobek druhých číslic a
- $c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0)$ je násobek součtů číslic a a b minus číslice c_2 a c_0.

Násobení velkých celých čísel – rozděl a panuj

Mějme dvě n-ciferná čísla a a b, kde n je sudé přirozené číslo.

Označíme první polovinu číslic čísla a jako a_1, druhou polovinu jako a_0. Zápis $a = a_1 a_0$ budeme chápat jako

$$a = a_1 a_0 = a_1 \cdot 10^{n/2} + a_0$$

Pro $b = b_1 b_0$ platí obdobný vztah.

Násobení velkých celých čísel – rozděl a panuj

Součin $c = a \times b$ můžeme zapsat jako

$$\begin{aligned} c &= \left(a_1 \cdot 10^{n/2} + a_0\right) \times \left(b_1 \cdot 10^{n/2} + b_0\right) \\ &= (a_1 \times b_1) \cdot 10^n + (a_1 \times b_0 + a_0 \times b_1) \cdot 10^{n/2} + (a_0 \times b_0) \\ &= c_2 \cdot 10^n + c_1 \cdot 10^{n/2} + c_0, \end{aligned}$$

kde

- $c_2 = a_1 \times b_1$ je násobek prvních polovin,
- $c_0 = a_0 \times b_0$ je násobek druhých polovin a
- $c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0)$ je násobek součtů polovin čísel a a b minus součet c_2 a c_0.

Čísla c_2, c_1 a c_0 vypočteme stejným způsobem – rekurzivní algoritmus.

Ukončení rekurze: $n = 1$ nebo čísla a, b lze násobit pomocí HW.

Násobení velkých celých čísel – počet násobení

- Počet násobení nutných pro výpočet součinu dvou n-ciferných čísel označíme jako $M(n)$.
- Výpočet součinu vyžaduje 3 násobení čísel poloviční velikosti. Násobení čísel pro $n = 1$ vyžaduje jedno násobení. Tedy

$$\begin{aligned} M(n) &= 3M\left(\frac{n}{2}\right) \text{ pro } n > 1 \\ M(1) &= 1 \end{aligned}$$

Násobení velkých celých čísel – počet násobení (pokrač.)

- Metodou zpětné substituce pro $n = 2^k$ dostáváme

$$\begin{aligned} M\left(2^k\right) &= 3M\left(2^{k-1}\right) = 3\left[3M\left(2^{k-2}\right)\right] = 3^2 M\left(2^{k-2}\right) \\ &\vdots \\ &= 3^i M(2^{k-i}) \\ &\vdots \\ &= 3^k M(2^{k-k}) = 3^k \end{aligned}$$

Násobení velkých celých čísel – počet násobení (pokrač.)

- Vzhledem k tomu, že $k = \log_2 n$ dostáváme dále

$$M(n) = 3^{\log_2 n} = n^{log_2 3} \approx n^{1,585}$$

Poznámky

1. Pro logaritmy platí $a^{\log_b c} = c^{\log_b a}$.
2. Rekurze nemusí nutně pokračovat až k $n = 1$, lze skončit dříve a pro malá n použít běžný algoritmus.

Násobení velkých celých čísel – počet sčítání a odčítání

- Jak je to ale se sčítáním a odčítáním? Není nižší počet násobení vykoupen vyšším počtem sčítání a násobení?
- Označme $A(n)$ počet sčítání a odčítání při násobení dvou n ciferných čísel.
- Kromě $3A\left(\frac{n}{2}\right)$ operací nutných pro rekurzivní výpočet c_2, c_1 a c_0 potřebujeme 5 součtů a 1 odečítání (na slajdu 471 označena barevně), tedy

$$\begin{aligned} A(n) &= 3A\left(\frac{n}{2}\right) + cn \text{ pro } n > 1 \\ A(1) &= 1 \end{aligned}$$

Násobení velkých celých čísel – počet sčítání a odčítání (pokrač.)

- Podle vztahu (**??**), **Master theorem**, dostáváme

$$A(n) \in \Theta\left(n^{\log_2 3}\right)$$

- Celkový počet sčítání a odčítání roste asymptoticky stejnou rychlostí jako počet násobení.

Násobení velkých celých čísel – historie

- Autorem algoritmu je sovětský matematik Anatolij Alexejevič Karacuba (1937 – 2008), který jej představil v roce 1960.
- Do té doby převládal názor, že tradiční algoritmus je asymptoticky optimální.
- Takže má smysl se zabývat i již „vyřešenými“ problémy :-)
- Otázkou je kdy použít standardní algoritmus a kdy algoritmus založený na strategii rozděl a panuj.

Strategie řešení rozděl a panuj

Strassenovo násobení matic

Strassenovo násobení matic

- Je násobení matic pomocí strategie hrubou silou nejlepší možné?
- Složitost násobení hrubou silou je $\Theta(n^3)$.
- Asymptoticky lepší algoritmus představil Volker Strassen v roce 1969.
- Výchozí „objev“ – násobení čtvercových matix řádu 2 lze provést se 7 násobeními, na rozdíl od 8 u hrubé síly.

Strassenovo násobení matic řádu 2

$$\begin{pmatrix} c_{0,0} & c_{0,1} \\ c_{1,0} & c_{1,1} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \times \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

$$\begin{aligned} m_1 &= (a_{0,0} + a_{1,1})(b_{0,0} + b_{1,1}) \\ m_2 &= (a_{1,0} + a_{1,1})b_{0,0} \\ m_3 &= a_{0,0}(b_{0,1} - b_{1,1}) \\ m_4 &= a_{1,1}(b_{1,0} - b_{0,0}) \end{aligned} \qquad \begin{aligned} m_5 &= (a_{0,0} + a_{0,1})b_{1,1} \\ m_6 &= (a_{1,0} - a_{0,0})(b_{0,0} + b_{0,1}) \\ m_7 &= (a_{0,1} - a_{1,1})(b_{1,0} + b_{1,1}) \end{aligned}$$

Strassenovo násobení matic

- Počty operací pro matice 2×2:

	Hrubá síla	Strassen
Počet násobení	8	7
Počet sčítání a odčítání	4	18

- Násobit takto matice 2×2 je očividný nesmysl. Ale!

Strassenovo násobení matic (pokrač.)

- Vztahy můžeme přeformulovat na převod násobení matic $n \times n$ na podmatice řádu $\frac{n}{2} \times \frac{n}{2}$ takto:

$$\left(\begin{array}{c|c} C_{0,0} & C_{0,1} \\ \hline C_{1,0} & C_{1,1} \end{array}\right) = \left(\begin{array}{c|c} A_{0,0} & A_{0,1} \\ \hline A_{1,0} & A_{1,1} \end{array}\right) \times \left(\begin{array}{c|c} B_{0,0} & B_{0,1} \\ \hline B_{1,0} & B_{1,1} \end{array}\right)$$

- Podmatici $C_{0,0}$ můžeme vypočítat buď jako

$$C_{0,0} = A_{0,0} \times B_{0,0} + A_{0,1} \times B_{1,0}$$

nebo jako

$$C_{0,0} = M_1 + M_4 - M_5 + M_7$$

- Matice $M_1, \ldots M_7$ vypočteme stejným, rekurzivním, způsobem.

Strassenovo násobení matic – analýza složitosti

Počet násobení $M(n)$ pro matice $n \times n$ je dán rekurentní rovnicí:

$$\begin{aligned} M(n) &= 7M\left(\frac{n}{2}\right) \text{ pro } n > 1 \\ M(1) &= 1 \end{aligned}$$

Předpokládejme, že $n = 2^k$ a odtud dostáváme

$$\begin{aligned} M\left(2^k\right) &= 7M\left(2^{k-1}\right) = 7\left[7M\left(2^{k-2}\right)\right] = 7^2 M\left(2^{k-2}\right) \\ &\vdots \\ &= 7^i M\left(2^{k-i}\right) \\ &\vdots \\ &= 7^k M(2^{k-k}) = 7^k. \end{aligned}$$

Protože $k = \log_2 n$ a tudíž

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807} < n^3$$

Strassenovo násobení matic – analýza složitosti, sčítání

- Neroste ale počet sčítání $A(n)$ pro matice $n \times n$ příliš rychle?
- Pro násobení matic $n \times n$ potřebujeme:
  1. vypočítat 7 podmatic řádu $\frac{n}{2} \times \frac{n}{2}$ a
  2. provést 18 sčítání/odečítání podmatic řádu $\frac{n}{2} \times \frac{n}{2}$.

  Takže

$$\begin{aligned} A(n) &= 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \quad \text{pro } n > 1 \\ A(1) &= 0 \end{aligned}$$

- Podle vztahu (**??**), **Master theorem**, dostáváme

$$A(n) \in \Theta\left(n^{\log_2 7}\right)$$

- Z toho plyne, že Strassenovo násobení matic má asymptotickou složitost $\Theta\left(n^{\log_2 7}\right)$, což je méně než řešení hrubou silou.

Strategie řešení rozděl a panuj

Problém nejbližší dvojice bodů

Strategie řešení rozděl a panuj

Konvexní obal množiny

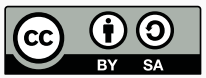
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Úvodní přednáška předmětu

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Úvodní přednáška předmětu

O předmětu Algoritmy II

O předmětu Algoritmy II

Upozornění

Všechny aktuální informace k předmětu naleznete na

http://www.cs.vsb.cz/dvorsky/

Tato prezentace slouží jen pro účely úvodní přednášky a nebude dále aktualizována.

O předmětu Algoritmy II

- Pokračování předmětu Algoritmy I, tj. náplní předmětu jsou další strategie algoritmického řešení úloh (dynamické programování, hladové algoritmy atd.) a typické příklady jejich užití.
- Přednášky – zaměřeny na **teorii**.
- Cvičení – zaměřena na **implementaci** řešení problémů danou strategií v jazyce C resp. C++.
- Vazby na další předměty:
  - Úvod do programování – jazyk C,
  - Funkcionální programování – rekurze a
  - Objektově orientované programování.

Rozsah předmětu, způsob zakončení

Rozsah předmětu

- výuka probíhá v zimním semestru druhého ročníku bakalářského studia
- hodinová dotace:
  - 2 hodiny přednášky a 2 hodiny cvičení týdně v prezenční formě a
  - 6 tutoriálů v kombinované formě studia
- předmět je ohodnocen 5 kredity

Způsob zakončení

- zakončení klasifikovaným zápočtem
- klasifikovaný zápočet není zkouška
- nejsou tři termíny jako u zkoušky, jiné dělení bodů v rámci celého předmětu

Garant předmětu, přednášející, tutor komb. formy

doc. Mgr. Jiří Dvorský, Ph.D.

Kancelář: EA441

Email: *jiri.dvorsky@vsb.cz*

Web: *www.cs.vsb.cz/dvorsky*

K čemu je garant předmětu?

Garant předmětu zodpovídá za průběh výuky celého předmětu, průběh cvičení, plnění úkolů na cvičeních a za korektní hodnocení úkolů. Problémy spojené se cvičeními řešte primárně se svým cvičícím. Nepodaří-li dosáhnout řešení problému s cvičícím obracejte se na garanta předmětu.

Prerekvizity

Povinná prerekvizita – úspěšné absolvování Algoritmů I.

Poznámka

- Algoritmy II jsou, jakožto povinný předmět, automaticky zapsány všem studentům.
- Nesplnění prerekvizity ale způsobí, že do Edisonu nejde zapsat jakýkoliv výsledek.
- Dokud nemáte splněné povinné prerekvizity, tak nemůžete absolvovat předmět, který je na nich závislý.

Docházka

Přednášky

- jsou obecně nepovinné
- účast na nich je doporučená

Cvičení

- jsou naopak povinná,
- účast a aktivita na cvičeních jsou hodnoceny,
- je nutno získat dostatečné bodové hodnocení.

Studenti se specifickými nároky

Centrum Slunečnice FEI

- *http://slunecnice-fei.vsb.cz/*,
- poskytuje podporu zpřístupňující studium i pro studenty se specifickými nároky,
- lze získat, mimo jiné, zvýšenou časovou dotaci na úkoly.

Výzva

- Pokud máte jakoukoliv úlevu, neprodleně kontaktujte svého cvičícího a garanta předmětu.
- Vyučující **nemají** k dispozici žádné seznamy studentů zahrnutých do tohoto programu.

Individuální studijní plán

- Individuální studijní plán umožňuje, v odůvodněných případech, individuální termíny pro plnění studijních povinností.
- Studenti, kteří mají individuální studijní plán, prosím neprodleně kontaktujte svého cvičícího a garanta předmětu a dohodněte se na individuálních termínech plnění úkolů.
- Individuální studijní plán neznamená naprostou libovůli v termínech.

Konzultační hodiny

- Pokud na přednášce nebudete něčemu rozumět, potřebujete poradit nebo vyřešit nějaký problém s přednáškou, cvičeními, testy, Vaší absencí na výuce atd. je možné využít **konzultační hodiny**.
- V tento čas jsem připraven věnovat se Vám osobně.
- Termín konzultačních hodin je uveřejněn mým webových stránkách.
- Žádám Vás ale o dodržení několika pravidel:
  1. Konzultaci je nutné si domluvit předem, nejlépe emailem.
  2. Pokud potřebujete poradit s učivem, přineste si s sebou materiály, které jste si k tématu prostudovali, vypište si co je Vám jasné a kde jste se „zasekli“ a potřebujete poradit.

Konzultační hodiny (pokrač.)

- Konzultací s vyučujícím nic neriskujete – maximálně se dozvíte co potřebujete.
- Přijďte se zeptat rovnou ke zdroji informací – internetová fóra jsou zaplevelena různými polopravdami i naprostými nesmysly.

Online konzultace

- Další možná forma konzultací.
- Určeno především pro kombinovanou formu studia.
- Termín konzultací je nutné individuálně dohodnout.
- Konzultace budou probíhat v MS Teams.

Algoritmy II – výuka, úkoly a jejich hodnocení pro různé formy studia

- Prezenční a kombinované studium má specifickou formu výuky.
- Obě formy studia mají specifické podmínky pro splnění předmětu.
- Podle formy Vašeho studia se Vás týká pouze jedna ze dvou následujících částí prezentace.

Úvodní přednáška předmětu

Prezenční forma studia

Témata přednášek

1. Úvodní přednáška předmětu
2. Strategie řešení transformuj a vyřeš
3. Záměna paměťové a časové složitosti
4. Dynamické programování
5. Hladové algoritmy
6. Strategie řešení iterativním zlepšováním
7. Meze možností algoritmického řešení problémů. P, NP a NP-úplné problémy.
8. Zdolávání mezí možností algoritmického řešení problémů

Cvičení

- Přímá výuka ve cvičeních odpovídá přednáškám.
- Ve cvičeních pracují studenti pod vedením cvičícího na konkrétní implementaci příkladů v jazyce C++.
- Dále je také možné konzultovat s cvičícím probírané učivo.
- Rozdělení do cvičení tak, jak je uvedeno v informačním systému Edison, je nutné respektovat.
- Není možné překračovat kapacitu cvičení.
- Veškeré přesuny je nutné mít zaznamenány v systému Edison.

Cvičení (pokrač.)

- Cvičení nenahrazuje přednášku!
  - Účelem cvičení není příprava na závěrečnou písemku.
  - Cvičení nejsou bleskovou přednáškou pro ty, kteří nechodí na přednášky.
  - Na cvičení je nutné být připraven.

Úkoly

- Hodnocení v předmětu Algoritmy II se skládá ze tří částí, úkolů:
  1. **průběžné aktivity na cvičeních**,
  2. **obhajoby projektu** a
  3. **závěrečné písemné práce**.
- Všechny úkoly jsou povinné.
- Z každého úkolu je nutné získat aspoň minimální počet bodů.

Úkoly – průběžná aktivita na cvičeních

- Tato část hodnocení probíhá **průběžně po celý semestr**.
- Na každém cvičení bude cvičícím ohodnocena Vaše aktivita. Aktivita je hodnocena pomocí barevného kódu:
  - zelená – student na cvičení pracoval aktivně, v látce se orientoval, dařilo se mu implementovat zadané úkoly,
  - oranžová – student na cvičení byl spíše pasivní, na cvičení nebyl příliš připraven (ve znalostech měl „mezery“), implementace úkolů se příliš nedařila a
  - červená – student na cvičení byl zcela pasivní, o výuku nejevil zájem, implementaci úkolů nezvládl. Do této kategorie spadá i neomluvená neúčast na cvičení.

Úkoly – průběžná aktivita na cvičeních (pokrač.)

- Každému barevnému kódu odpovídá určitá váha, která se projeví v celkovém hodnocení všech cvičení. Zelená aktivita má váhu 1, oranžová má váhu 0,5 a červená 0.
- Z takto získaných vah z jednotlivých cvičení se na konci semestru vypočte průměrná váha, která se vynásobí maximálním možným počtem bodů (30) a výsledek je Vámi získaný počet bodů.
- Je zřejmé, že všechny zelené kódy odpovídají maximálnímu počtu bodů (30), samé červené kódy odpovídají nulovému počtu bodů.
- Body za aktivitu nelze získat zpětně.

Úkoly – průběžná aktivita na cvičeních (pokrač.)

Příklad

Student Franta na pěti cvičeních získal zelené hodnocení, na třech oranžové a na dvou červené. Průměrnou váhu vypočtene jako:

$$\frac{5 \times 1 + 3 \times 0,5 + 2 \times 0}{5 + 3 + 2} = \frac{6,5}{10} = 0,65.$$

Výsledné bodové hodnocení je tedy $0,65 \times 30 = 19,5$ bodů.

Úkoly – obhajoba projektu

- Zadání projektu bude zveřejněno na webu předmětu koncem října.
- Deadline odevzdání je 10. prosince 2023 23:59.
- Způsob odevzdání stanoví jednotliví cvičící.
- Obhajoby projektů proběhnou v zápočtovém týdnu a ve zkouškovém období. Harmonogram obhajob je v kompetenci cvičících.
- Bez ohledu na to, kdy proběhnou obhajoby projektů platí, že se obhajuje verze, která byla odevzdána do deadlinu.
- Na obhajobu projektu není možný opravný termín.

Úkoly – závěrečná písemná práce

- Závěrečná písemná práce proběhne ve zkouškovém období.
- Termíny budou zveřejněny v systému Edison.
- Opravný termín na závěrečnou písemnou práci je poskytován jen těm studentům, kteří u svého prvního pokusu získali aspoň 10 bodů.

Počet bodů na prvním termínu	Opravný termín
0 až 9	NE
10 až 20	ANO
více než 21	není nutný, úspěch

Úkoly – závěrečná písemná práce (pokrač.)

- Závěrečnou písemnou práci máte možnost psát celkem **dvakrát**, jinak řečeno máte nárok na jednu opravu. Předmět je ukončen klasifikovaným zápočtem. Nevztahuje se tudíž na něj požadavek dvou oprav, jak to vyžaduje studijní řád u zkoušky.
- Žádné další opravy nejsou možné.

Hodnocení úkolů

- Je nutné splnit **všechny výše uvedené úkoly**,
- a zároveň u všech úkolů **aspoň minimální počet bodů**.

Úkol	Počet bodů	
	minimum	maximum
Průb. aktivita na cvičeních	15	30
Obhajoba projektu	15	30
Písemná práce	21	40
Celkový počet bodů	51	100

Obecné pokyny ke všem úkolům

- U všech úkolů jste povinni se prokázat svou studentskou kartou nebo jiným oficiálním dokladem totožnosti. Bez prokázání totožnosti Vám nebude výsledek započítán.
- Každý prohřešek vůči studijnímu řádu u testů a písemné práce bude nekompromisně postihován. Jde především o opisování, plagiátorství, a záměnu studentů.
- Je zakázáno zadání testů, písemek atd. kopírovat, fotit mobily, fotoaparáty, skenovat či jakkoliv jinak kopírovat, rozmnožovat, sdílet elektronickým způsobem a podobně.

Úvodní přednáška předmětu

Kombinovaná forma studia

Tutoriály

1. tutoriál – 22. září 2023 povinný

- Na tomto úvodním tutoriálu Vám budou sděleny informace o organizaci studia předmětu a informace o náplni předmětu.
- Konzultace k tématům: Strategie řešení transformuj a vyřeš.

2. tutoriál – 7. října 2023

- Konzultace k tématům: Záměna paměťové a časové složitosti.

Tutoriály (pokrač.)

3. tutoriál – 20. října 2023

- Konzultace k tématům: Dynamické programování.

4. tutoriál – 3. listopadu 2023

- Konzultace k tématům: Hladové algoritmy.

Tutoriály (pokrač.)

5. tutoriál – 24. listopadu 2023

- Konzultace k tématům: Strategie řešení iterativním zlepšováním.

6. tutoriál – 8. prosince 2023

- Konzultace k tématům: Meze možností algoritmického řešení problémů. P, NP a NP-úplné problémy. Zdolávání mezí možností algoritmického řešení problémů.

Úkoly

- Hodnocení v předmětu Algoritmy II se skládá ze tří částí, úkolů:
  1. **průběžné aktivity na tutoriálech**,
  2. **obhajoby projektu** a
  3. **závěrečné písemné práce**.
- Všechny úkoly jsou povinné.
- Z každého úkolu je nutné získat aspoň minimální počet bodů.
- Další informace o jednotlivých úkolech budou k dispozici na webu tutora.

Úkoly – průběžná aktivita na tutoriálech

Průběžná aktivita na tutoriálech znamená:

- účast na tutoriálech a
- průběžné plnění úkolů zadaných na jednotlivých tutoriálech.

Úkoly – obhajoba projektu

- Zadání projektu bude zveřejněno na webu předmětu koncem října.
- Deadline odevzdání je 10. prosince 2023 23:59.
- Způsob odevzdání a další náležitosti budou upřesněny na webu tutora.
- Obhajoby projektů proběhnou ve zkouškovém období. Termíny budou vypsány v systému Edison.
- Bez ohledu na to, kdy proběhnou obhajoby projektů platí, že se obhajuje verze, která byla odevzdána do deadlinu.
- Na obhajobu projektu není možný opravný termín.

Úkoly – závěrečná písemná práce

- Závěrečná písemná práce je zaměřena na teoretické znalosti.
- Závěrečné písemná práce proběhne ve zkouškovém období.
- Termíny budou vypsány v systému Edison.
- Opravný termín na závěrečnou písemnou práci je poskytován jen těm studentům, kteří u svého prvního pokusu získali aspoň 10 bodů.

Počet bodů na prvním termínu	Opravný termín
0 až 9	NE
10 až 20	ANO
více než 21	není nutný, úspěch

Úkoly – závěrečná písemná práce (pokrač.)

- Závěrečnou písemnou práci máte možnost psát celkem **dvakrát**, jinak řečeno máte nárok na jednu opravu. Předmět je ukončen klasifikovaným zápočtem. Nevztahuje se tudíž na něj požadavek dvou oprav, jak to vyžaduje studijní řád u zkoušky.
- Žádné další opravy nejsou možné.

Hodnocení úkolů

- Je nutné splnit **všechny výše uvedené úkoly**,
- a zároveň u všech úkolů **aspoň minimální počet bodů**.

Úkol	Počet bodů	
	minimum	maximum
Průb. aktivita na tutoriálech	15	30
Obhajoba projektu	15	30
Písemná práce	21	40
Celkový počet bodů	51	100

Obecné pokyny ke všem úkolům

- U všech úkolů jste povinni se prokázat svou studentskou kartou nebo jiným oficiálním dokladem totožnosti. Bez prokázání totožnosti Vám nebude výsledek započítán.
- Každý prohřešek vůči studijnímu řádu u testů a písemné práce bude nekompromisně postihován. Jde především o opisování, plagiátorství, a záměnu studentů.
- Je zakázáno zadání testů, písemek atd. kopírovat, fotit mobily, fotoaparáty, skenovat či jakkoliv jinak kopírovat, rozmnožovat, sdílet elektronickým způsobem a podobně.

Úvodní přednáška předmětu

Software pro výuku

Software pro výuku

Primární software

- Vývojové prostředí pro C++
- Dokumentace k C++

Doplňkový software

- Dokumentační systém Doxygen, *www.doxygen.org*
- Typografický systém LaTeX, *www.ctan.org*

Vývojové prostředí pro C++

- Na učebnách je pro výuku k dispozici Microsoft Visual Studio Community 2022.
- Toto vývojové prostředí doporučuji i pro domácí přípravu.
- Obecně lze použít jakékoliv vývojové prostředí s kompilátorem podporujícím minimálně specifikaci **C++17**.

Vývojové prostředí pro C++ (pokrač.)

Poznámky

1. Při hodnocení Vašich projektů bude používán překladač **Microsoft Visual C++** a specifikace jazyka **C++17**.
2. Pozor na nestandardní rozšíření jazyka C++ implementovaného v GNU C++ kompilátoru.
3. Jazyk C není totožný s jazykem C++!

Úvodní přednáška předmětu

Studijní literatura

Studijní literatura

Studijní literaturu lze rozdělit do dvou skupin:

- **povinná literatura** – strategie algoritmického řešení problémů a
- **doporučená literatura** – programovací jazyk C++.

Níže uvedenou literaturu využijete v předmětu Algoritmy I i Algoritmy II.

Povinná literatura

1. LEVITIN, Anany. *Introduction to the Design and Analysis of Algorithms*. 3rd ed. Boston: Pearson, 2012. ISBN 978-0-13-231681-1.
2. CORMEN, Thomas H., Charles Eric LEISERSON, Ronald L. RIVEST a Clifford STEIN, [2022]. *Introduction to algorithms*. Fourth edition. Cambridge, Massachusetts: The MIT Press. ISBN 978-026-2046-305.
3. SEDGEWICK, Robert. *Algoritmy v C*. Praha: SoftPress, 2003. ISBN 80-864-9756-9.

Povinná literatura (pokrač.)

4. MAREŠ, Martin a Tomáš VALLA, 2017. *Průvodce labyrintem algoritmů* [online]. Praha: CZ.NIC, z.s.p.o. [cit. 2020-10-03]. CZ.NIC. ISBN 978-80-88168-19-5. Dostupné z: *https://knihy.nic.cz/*
5. WRÓBLEWSKI, Piotr. *Algoritmy*. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.
6. WIRTH, N. *Algoritmy a štruktúry údajov*. Alfa, Bratislava 1989.

Doporučená literatura

1. STROUSTRUP, Bjarne. *C++ programovací jazyk*. Praha: Softwarové Aplikace a Systémy, 1997. ISBN 80-901-5072-1.
2. VIRIUS, Miroslav. *Pasti a propasti jazyka C++*. 2., aktualiz. a rozš. vyd. Brno: CP Books, 2005. ISBN 80-251-0509-1.
3. SCHILDT, Herbert. *Nauč se sám C++: [poznej, vyzkoušej, používej]*. Praha: SoftPress, 2001. ISBN 80-864-9713-5.
4. ECKEL, Bruce. *Myslíme v jazyku C++*. Praha: Grada, 2000. Knihovna programátora (Grada). ISBN 80-247-9009-2.

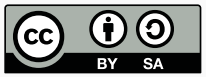
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Strategie řešení transformuj a vyřeš

doc. Mgr. Jiří Dvorský, Ph.D.

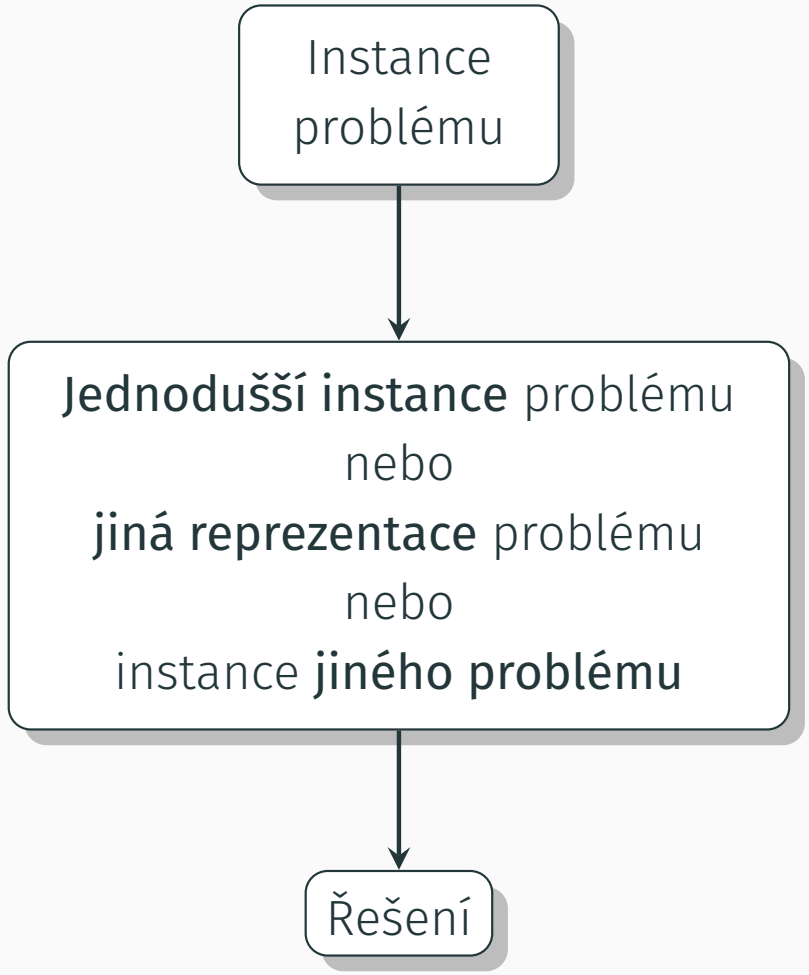
Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Strategie řešení transformuj a vyřeš

Dvoufázová strategie

1. transformace
2. řešení



Strategie řešení transformuj a vyřeš

Předtřídění dat

Předtřídění dat

- Poměrně stará myšlenka, která mimo jiné motivovala výzkum třídících algoritmů.
- Setříděná data vedou na výrazně jednodušší algoritmy, „pořádek musí být“.
- Předpoklady:
  1. data jsou uložena v poli – třídění pole je snazší než třídění seznamu
  2. pro třídění použijeme algoritmus se složitostí $\Theta(n \log n)$ – typicky QuickSort, MergeSort.
- Využití: geometrické algoritmy, grafové algoritmy, žravé algoritmy.

Jedinečnost prvků v poli

Zadání

Máme dáno pole A s n prvky. Máme určit, zda se v poli A vyskytuje každý prvek právě jednou.

Řešení hrubou silou – porovnáváme všechny dvojice prvků dokud:

1. nenajdeme dvojici stejných prvků nebo
2. jsme otestovali všechny dvojice prvků.

Časová složitost je v nejhorším případě $\Theta(n^2)$.

Jedinečnost prvků v poli

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

Časová složitost algoritmu

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Výpočet modu

Zadání

Máme dáno pole A s n prvky. Máme určit, který prvek se v poli vyskytuje nejčastěji. Tento prvek se nazývá **modus**.

Pro jednoduchost budeme předpokládat, že v poli A existuje jen jeden modus.

Řešení hrubou silou

Pro každý prvek $a_i \in A$ prohledáme pomocný seznam L:

1. pokud nalezneme shodu, inkrementujeme příslušnou četnost,
2. v opačném případě vložíme prvek a_i na konec seznamu s četností 1.

Výpočet modu – časová složitost řešení hrubou silou

- Nejhorší případ – všechny prvky v poli A jsou různé.
- Pro a_i musíme provést $i - 1$ porovnání s prvky v seznamu L, než přidáme nový prvek na jeho konec.
- Počet porovnání je tedy roven

$$C(n) = \sum_{i=1}^{n} (i - 1) = 0 + 1 + \cdots + (n - 1) = \frac{1}{2} n(n - 1) \in \Theta(n^2)$$

- Nalezení maxima vyžaduje $n - 1$ porovnání, což neovlivní kvadratickou složitost algoritmu.

Výpočet modu – předtřídění dat

- Pokud pole A setřídíme, budou shodné prvky v poli A vedle sebe.
- Pro výpočet modu stačí nalézt nejdelší úsek (angl. run) shodných prvků v A.
- Časová složitost

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Výpočet modu

ALGORITHM *PresortMode*$(A[0..n-1])$

//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode
sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \leq n-1$ **do**
    $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
    while $i + runlength \leq n-1$ **and** $A[i + runlength] = runvalue$
        $runlength \leftarrow runlength + 1$
    if $runlength > modefrequency$
        $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
    $i \leftarrow i + runlength$
return $modevalue$

Vyhledávání prvku x v poli A délky n

- Řešení hrubou silou vede na algoritmus vyžadující n porovnání v nejhorším případě.
- Po setřídění pole, lze použít algoritmus půlení intervalu, který vyžaduje $\lfloor \log_2 n \rfloor + 1$ porovnání v nejhorším případě.
- Časová složitost algoritmu potom bude

  $$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

  což je **více** než složitost sekvenčního vyhledávání!!!
- Ale pro **opakované** vyhledávání se již vyplatí pole A setřídit.

Zdroje pro samostatné studium

- Kniha [2], kapitola 6.1, strany 202 – 205

Strategie řešení transformuj a vyřeš

Gaussova eliminační metoda

Gaussova eliminační metoda – motivace

Soustavu dvou rovnic o dvou neznámých

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

lze řešit poměrně snadno – například proměnnou x vyjádříme jako funkci y, dosadíme do druhé rovnice a rovnici vyřešíme.

Problém

Jak řešit soustavu n rovnic o n neznámých? Stejným způsobem?

Gaussova eliminační metoda

Soustavu n lineárních rovnic o n neznámých

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

transformujeme na ekvivaletní soustavu rovnic, kde všechny koeficienty pod hlavní diagonálou jsou nulové

$$\begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{nn}x_n &= b'_n \end{aligned}$$

Gaussova eliminační metoda – maticový zápis

$$\mathbf{A}\vec{x} = \vec{b} \quad \Longrightarrow \quad \mathbf{A}'\vec{x} = \vec{b}'$$

kde

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{pmatrix}$$

$$\mathbf{A}' = \begin{pmatrix} a'_{11} & a_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{pmatrix} \qquad \vec{b}' = \begin{pmatrix} b'_{11} \\ b'_{21} \\ \vdots \\ b'_{n1} \end{pmatrix}$$

$\mathbf{A}'$ se nazývá **horní trojúhelníková matice**.

Gaussova eliminační metoda – výhody změny reprezentace

Soustavu danou horní trojúhelníkovou maticí lze snadno řešit pomocí **zpětné substituce**:

1. Z rovnice
$$a'_{nn} x_n = b'_n$$
vypočteme neznámou x_n.
2. Hodnotu neznámé x_n dosadíme do rovnice
$$a'_{n-1\,n-1} x_{n-1} + a'_{n-1\,n} x_n = b'_{n-1}$$
a vypočteme neznámou x_{n-1}.
3. Takto postupujeme dále až k výpočtu neznámé x_1.

Složitost tohoto algoritmu je $\Theta(n^2)$.

Gaussova eliminační metoda – elementární operace

Matici soustavy $\mathbf{A}$ převedeme na horní trojúhelníkovou matici $\mathbf{A}'$ pomocí **elementárních operací**:

- záměna dvou rovnic v soustavě,
- vynásobení rovnice nenulovým koeficientem a
- přičtení či odečtení násobku jiné rovnice k dané rovnici, tj. lineární kombinace s jinou rovnici.

Elementární operace nemění řešení soustavy rovnic – transformovaná soustava má stejné řešení jako původní soustava.

Gaussova eliminační metoda – transformace matice

1. Zvolíme a_{11} jako **pivot** a „vynulujeme“ všechny koeficienty v prvním sloupcí, kromě a_{11}.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

   „Vynulování“ – od druhé rovnice odečteme $\frac{a_{21}}{a_{11}}$ násobek první rovnice, od třetí rovnice odečteme $\frac{a_{31}}{a_{11}}$ násobek první rovnice,...
2. Zvolíme a_{22} jako pivot a opakujeme stejný postup.

Poznámka

Změny provádíme pochopitelně i pro vektor pravých stran $\vec{b}$.

Gaussova eliminační metoda – příklad

Mějme soustavu rovnic

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

Rozšířená matice soustavy

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right)$$

Gaussova eliminační metoda – příklad (pokrač.)

Dopředná eliminace

Od druhého řádku odečteme $\frac{4}{2}$ násobek prvního řádku, od třetího řádku odečteme $\frac{1}{2}$ násobek prvního řádku

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{3}{2} \end{array}\right)$$

Od třetího řádku odečtem $\frac{1}{2}$ násobek druhého řádku

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right)$$

Gaussova eliminační metoda – příklad (pokrač.)

Zpětná substituce

$$
\begin{aligned}
x_3 &= \frac{-2}{2} = -1 \\
x_2 &= \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0 \\
x_1 &= \frac{1 - x_3 - (-1)x_2}{2} = \frac{1 - (-1)}{2} = 1
\end{aligned}
$$

Gaussova eliminační metoda – dopředná eliminace

ALGORITHM *ForwardElimination*$(A[1..n, 1..n], b[1..n])$

//Applies Gaussian elimination to matrix A of a system's coefficients,
//augmented with vector b of the system's right-hand side values
//Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
//Output: An equivalent upper-triangular matrix in place of A with the
//corresponding right-hand side values in the $(n + 1)$st column
for $i \leftarrow 1$ **to** n **do** $A[i, n + 1] \leftarrow b[i]$ //augments the matrix
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        for $k \leftarrow i$ **to** $n + 1$ **do**
            $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] \,/\, A[i, i]$

Gaussova eliminační metoda – dopředná eliminace

Částečné pivotování

- V algoritmu dopředné eliminace je chyba. Pokud $a_{ii} = 0$, tak dojde k dělení nulou.
- Problém lze řešit výměnou rovnic (elementární operace) tak, aby $a_{ii} \neq 0$.
- Lze současně řešit i případné zaokrouhlovací chyby – pivot volíme tak, aby byl ze všech prvků a_{ii} až a_{ni} v absolutní hodnotě největší.

Opakované výpočty

V nejvnitřnějším cyklu se podíl $\frac{a_{ji}}{a_{ii}}$ počítá opakovaně – stačí jej spočítat jednou mimo cyklus.

Gaussova eliminační metoda – částečné pivotování

ALGORITHM *BetterForwardElimination*$(A[1..n, 1..n], b[1..n])$

//Implements Gaussian elimination with partial pivoting
//Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the $(n+1)$st column
for $i \leftarrow 1$ **to** n **do** $A[i, n+1] \leftarrow b[i]$ //appends b to A as the last column
for $i \leftarrow 1$ **to** $n-1$ **do**
    $pivotrow \leftarrow i$
    for $j \leftarrow i+1$ **to** n **do**
        if $|A[j, i]| > |A[pivotrow, i]|$ $pivotrow \leftarrow j$
    for $k \leftarrow i$ **to** $n+1$ **do**
        $swap(A[i, k], A[pivotrow, k])$
    for $j \leftarrow i+1$ **to** n **do**
        $temp \leftarrow A[j, i] \,/\, A[i, i]$
        for $k \leftarrow i$ **to** $n+1$ **do**
            $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$

Gaussova eliminační metoda – časová složitost

- Velikost vstupu – počet rovnic v soustavě, tj. rozměr matice n.
- Základní operace – aritmetické operace, z historických důvodů násobení. V nejvnitřnějším cyklu počet násobení odpovídá počtu odčítání, jde jen o násobek konstantou 2.
- Bude nás zajímat počet násobení $C(n)$ v závislosti na čísle n.

Gaussova eliminační metoda – časová složitost (pokrač.)

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+2-i) \\
&= \sum_{i=1}^{n-1} (n+2-i)[n-(i+1)+1] = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
&= (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1 \\
&= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} \\
&= \frac{n(n-1)(2n+5)}{6} = \frac{2n^3+3n^2-5n}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3)
\end{aligned}
$$

Gaussova eliminační metoda – časová složitost (pokrač.)

Protože složitost zpětné substituce je $\Theta(n^2)$, je složitost celé Gaussovy eliminační metody $\Theta(n^3)$.

LU-rozklad matice

Mějme matici $\mathbf{A}$ soustavy lineárních rovnic z předchozího příkladu

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

Dále uvažujme dvě matice:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}$$

Koeficienty z Gaussovy eliminace

$$\mathbf{U} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Výsledek Gaussovy eliminace

LU-rozklad matice

Definice

Mějme $\mathbf{A}$ regulární čtvercovou matici s prvky z $\mathbb{R}$, u které není třeba při Gaussově eliminaci prohazovat řádky. Pak existují regulární matice $\mathbf{L}$ a $\mathbf{U}$, které jsou určeny jednoznačně a platí pro ně následující tvrzení

$$\mathbf{A} = \mathbf{L}\mathbf{U},$$

kde $\mathbf{L}$ je dolní trojúhelníková matice s jedničkami na celé hlavní diagonále a $\mathbf{U}$ horní trojúhelníková matice s nenulovými prvky na hlavní diagonále.

Řešení soustavy rovnic *LU*-rozkladem

Mějme soustavu lineárních rovnic

$$\mathbf{A}\vec{x} = \vec{b}$$

Matici $\mathbf{A}$ nahradíme jejím *LU* rozkladem

$$\mathbf{L}\mathbf{U}\vec{x} = \vec{b}$$

Dále označme součin $\mathbf{U}\vec{x} = \vec{y}$. Po dosazení dostáváme soustavu rovnic

$$\mathbf{L}\vec{y} = \vec{b}$$

Tuto soustavu můžeme snadno vyřešit, protože $\mathbf{L}$ je dolní trojúhelníková matice. A nakonec můžeme lehce vyřešit i soustavu

$$\mathbf{U}\vec{x} = \vec{y},$$

protože $\mathbf{U}$ je horní trojúhelníková matice.

Řešení soustavy rovnic *LU*-rozkladem, příklad

Mějme soustavu rovnic

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

Provedeme ***LU***-rozklad matice soustavy $\mathbf{A}$

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Řešení soustavy rovnic *LU*-rozkladem, příklad (pokrač.)

Nejprve budeme řešit soustavu $\mathbf{L}\vec{y} = \vec{b}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ 0 \end{pmatrix}$$

$$\begin{aligned} y_1 &= 1 \\ y_2 &= 5 - 2y_1 = 3 \\ y_3 &= 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2 \end{aligned}$$

Řešení soustavy rovnic *LU*-rozkladem, příklad (pokrač.)

Následně vyřešíme soustavu $\mathbf{U}\vec{x} = \vec{y}$

$$\begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}$$

$$\begin{aligned} x_3 &= \frac{-2}{2} = -1 \\ x_2 &= \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0 \\ x_1 &= \frac{1 - x_3 - (-1)x_2}{2} = \frac{1 - (-1)}{2} = 1 \end{aligned}$$

LU-rozklad matice, poznámky

- V praxi se pro řešení soustav lineárních rovnic používá právě *LU*-rozklad.
- Pomocí *LU*-rozkladu lze efektivně řešit více soustav rovnic se shodnou maticí soustavy.
- Matice $\mathbf{L}$ a $\mathbf{U}$ lze uložit společně v jedné „matici“ – z matice $\mathbf{L}$ ukládáme jen prvky pod diagonálou. Proč?

Inverzní matice

Definice

Ke každé regulární čtvercové matici $\mathbf{A}$ řádu n existuje právě jedna čtvercová matice $\mathbf{A}^{-1}$ řádu n taková, že

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}.$$

Matice $\mathbf{A}^{-1}$ se nazývá **inverzní matice** k matici $\mathbf{A}$.

Matice bez inverze se nazývají **singulární**.

Inverzní matice – obdoba převráceného čísla u racionálních či reálných čísel; srovnej řešení lineární rovnice a soustavy lineárních rovnic

$$ax = b \qquad\qquad \mathbf{A}\vec{x} = \vec{b}$$

$$x = \frac{1}{a}b = a^{-1}b \qquad\qquad \vec{x} = \mathbf{A}^{-1}\vec{b}$$

Výpočet inverzní matice

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

Musíme řešit n soustav lineárních rovnic o n neznámých se stejnou maticí $\mathbf{A}$

$$\mathbf{A}\vec{x}^j = \vec{e}^j,$$

kde $\vec{x}^j$ resp. $\vec{e}^j$ je j-tý sloupcový vektor matice $\mathbf{A}$ resp. $\mathbf{I}$.

Využijeme LU rozklad

$$\mathbf{LU}\vec{x}^j = \vec{e}^j$$

Determinant matice

Definice

Determinant čtvercové matice $\mathbf{A}$ řádu n definujeme předpisem

$$\det \mathbf{A} = \sum_{p \in P_n} \sigma(p) a_{1p_1} a_{2p_2} \cdots a_{np_n},$$

kde

- P_n je množina všech permutací čísel $\{1, \ldots, n\}$ a
- $\sigma(p) = (-1)^s$ je znaménko permutace, s označuje počet inverzí v permutaci p.

Determinant $\det \mathbf{A}$ zapisujeme i jako $|\mathbf{A}|$.

Výpočet determinantů matic řádu 1, 2 a 3

$$\begin{vmatrix} a_{11} \end{vmatrix} = a_{11}$$

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{13}a_{21}a_{32} + a_{12}a_{23}a_{31} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}$$

Poslední vzorec je znám také jako tzv. Sarrusovo pravidlo.

Výpočet determinantů matic vyšších řádů

- Výpočet podle definice vyžaduje sečíst $n!$ součinů prvků matice.
- Opět lze využít, lehce modifikovanou, Gaussovu eliminaci a upravit matici na horní trojúhelníkovou. Platí věta, že determinant horní (dolní) trojúhelníkové matice je roven součinu prvků na její diagonále.
- Determinanty vyšších řádů tak, lze počítat s kubickou časovou složitostí.

Výpočet determinantů matic vyšších řádů – příklad

$$\begin{vmatrix} 3 & 6 & -9 \\ 4 & 8 & 10 \\ 2 & 7 & 5 \end{vmatrix} = 3 \begin{vmatrix} 1 & 2 & -3 \\ 4 & 8 & 10 \\ 2 & 7 & 5 \end{vmatrix} = 3 \begin{vmatrix} 1 & 2 & -3 \\ 0 & 0 & 22 \\ 0 & 3 & 11 \end{vmatrix} = -3 \begin{vmatrix} 1 & 2 & -3 \\ 0 & 3 & 11 \\ 0 & 0 & 22 \end{vmatrix}$$

$$= -3 \cdot 66 = -198$$

Poznámka

Tento výpočet slouží pouze pro ukázku, determinant matice řádu 3 lze pochopitelně počítat přímo vzorcem.

Cramerovo pravidlo

Řešení soustavy rovnic $\mathbf{A}\vec{x} = \vec{b}$ lze vyjádřit jako

$$\vec{x} = \left(\frac{\det \mathbf{A}_1}{\det \mathbf{A}}, \dots, \frac{\det \mathbf{A}_i}{\det \mathbf{A}}, \dots, \frac{\det \mathbf{A}_n}{\det \mathbf{A}}, \right)$$

kde $\mathbf{A}_i$ je matice, která vznikne záměnou i-tého sloupce matice $\mathbf{A}$ za vektor pravých stran $\vec{b}$.

Složitost algoritmu

- Výpočet determinantu řádu n lze zvládnout v čase $\Theta(n^3)$.
- Musíme vypočítat n determinantů matic $\mathbf{A}_i$ a jeden determinant matice $\mathbf{A}$, celkem tedy $n + 1$ determinantů.
- Celková složitost výpočtu řešení soustavy rovnic Cramerovým pravidlem má složitost $\Theta(n^4)$.

Zdroje pro samostatné studium

- Kniha [2], kapitola 6.2, strany 208 – 216
- Kniha [3], kapitoly 28.1 a 28.2, strany 819 – 838
- Kniha [5], kapitoly 12 a 13, strany 133 – 163
- Kniha [6], kapitoly 1.3 a 1.4, strany 24 – 36

Strategie řešení transformuj a vyřeš

Vyvážené vyhledávací stromy

Binární vyhledávací stromy – připomenutí

- Fundamentální datová struktura pro implementaci množin, slovníků atd.
- Každý uzel obsahuje jeden klíč; nad klíči musí být definováno uspořádání.
- Pro každý uzel platí, že všechny klíče v levém podstromu jsou menší než klíč v daném uzlu a v pravém podstromu jsou všechny klíče větší.
- Průměrná časová složitost hledání, vkládání a mazání uzlů je $\Theta(\log_2 n)$.
- Nejhorší případ je ale stále $\Theta(n)$ – strom degeneruje na seznam.

Vyvážené vyhledávací stromy

Možná řešení nejhoršího případu:

Aktivní opatření

- transformace na vyvážený binární strom pomocí rotací
- různé definice vyváženosti
- AVL stromy, červeno-černé stromy, splay stromy.

Změna reprezentace

- více klíčů v jednom uzlu,
- 2-3 stromy, 2-3-4 stromy, B-stromy.

AVL stromy

Autoři

- Georgij Maximovič **A**delson-**V**elskij a
- Jevgenij Michajlovič **L**andis

Poprvé publikováno v roce 1962.

Definice

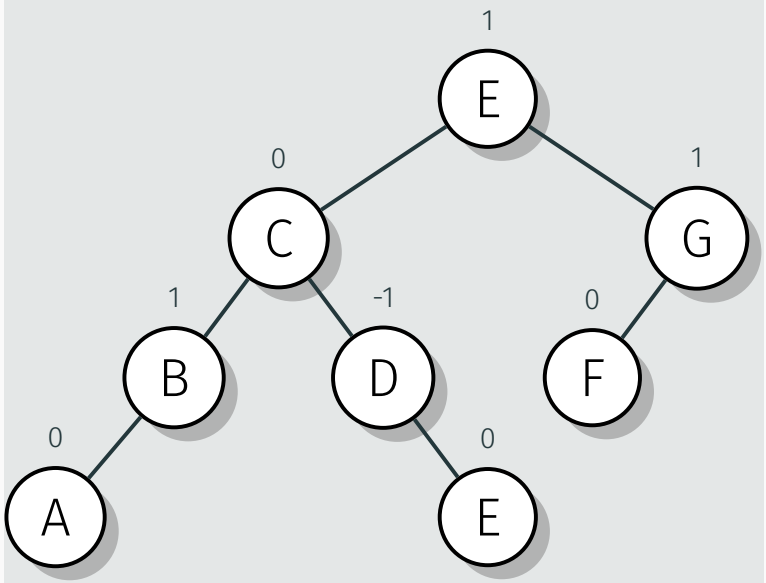
Faktorem vyváženosti uzlu u nazýváme rozdíl výšek jeho levého a pravého podstromu. Výšku prázdného stromu definujeme jako -1.

Definice

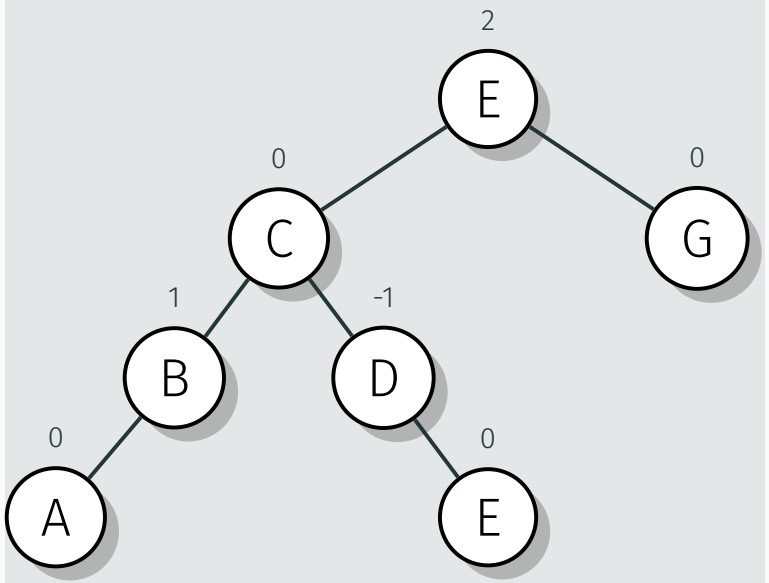
Binární vyhledávací strom nazýváme **AVL stromem** tehdy a jen tehdy, je-li faktor vyváženosti pro každý uzel ve stromu buď -1, 0 nebo +1.

AVL stromy – příklad

AVL strom



Toto není AVL strom

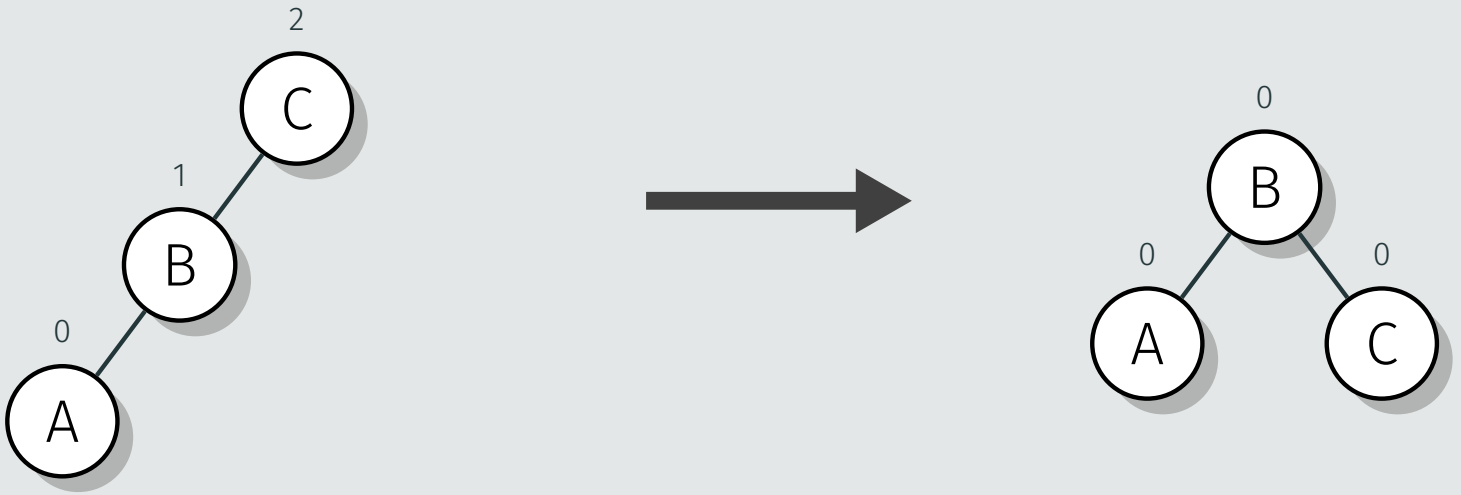


AVL stromy – udržování vyváženosti

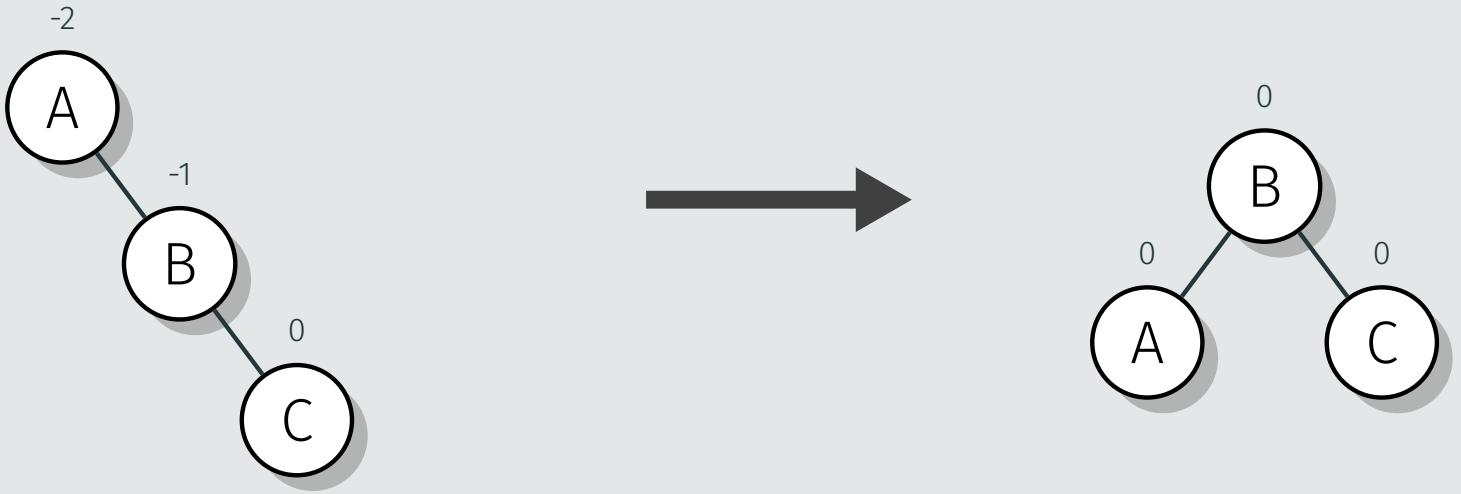
- Vložení nového uzlu, resp. smazání exstujícího, může způsobit vyváženost AVL stromu.
- Vyváženost je nutné po každé takové operaci obnovit.
- Vyvíženost se obnovuje pomocí **rotací**.
- Rotace je lokální transformace stromu v těch uzlech, kde faktor vyváženosti dosáhne hodnoty -2 nebo 2.
- Pokud je takových uzlů více, začínáme vždy uzlem na nejnižší úrovni (co nejblíže k listům stromu).
  A postupujeme vzhůru ke kořeni stromu.
- Existují celkem čtyři rotace – dvě dvojice navzájem zrcadlově symetrických rotací.

Jednoduché rotace

R-rotace

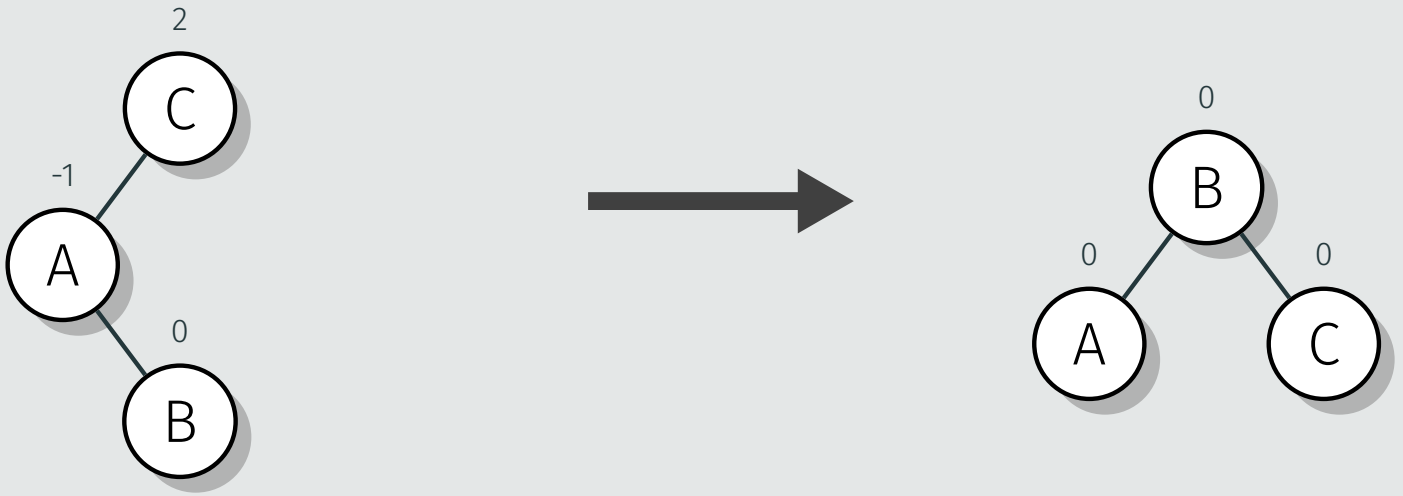


L-rotace

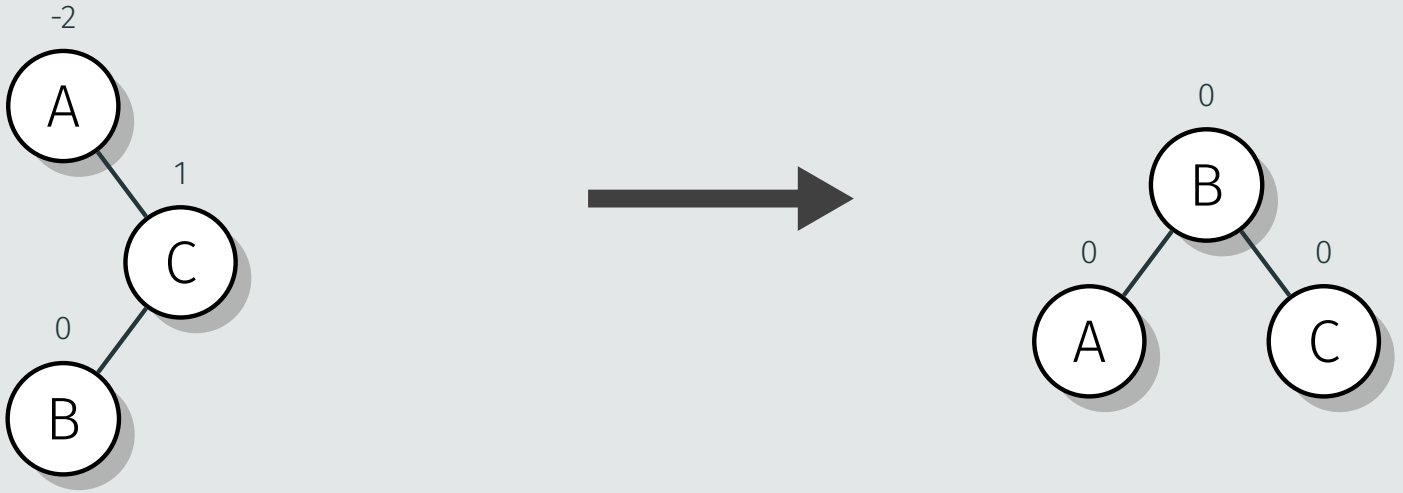


Dvojité rotace

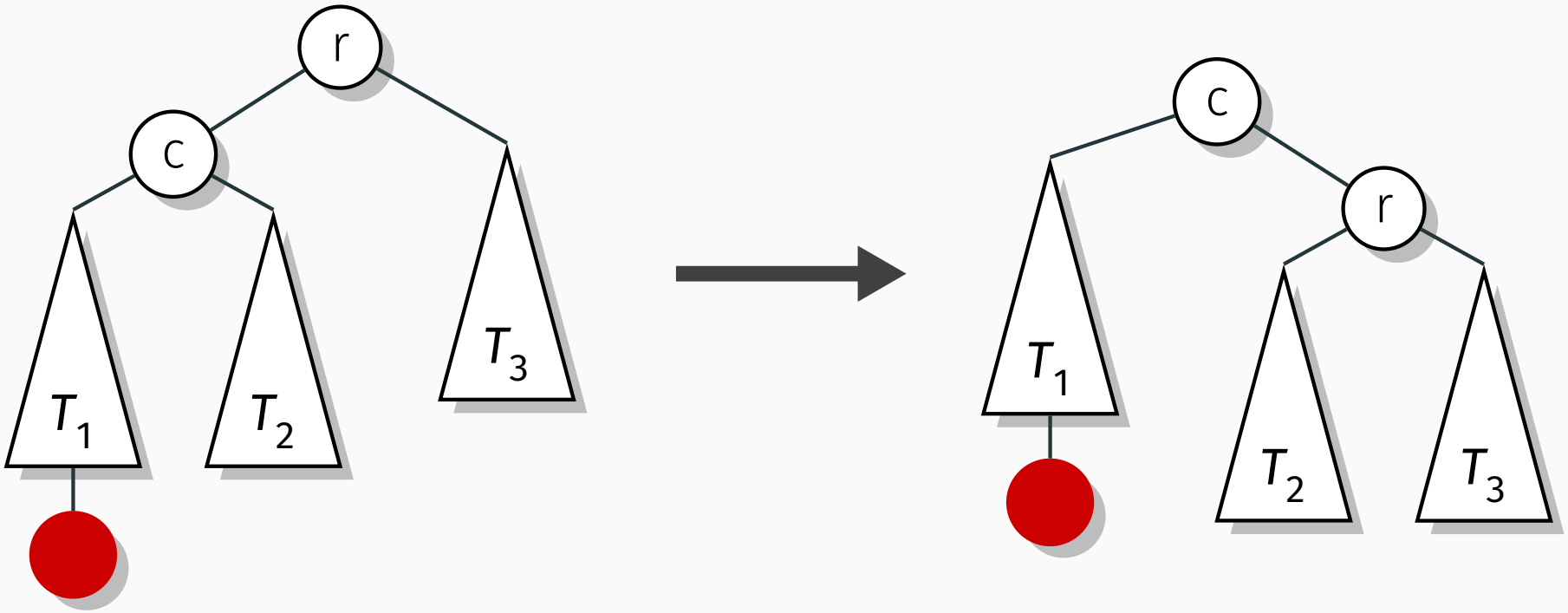
LR-rotace



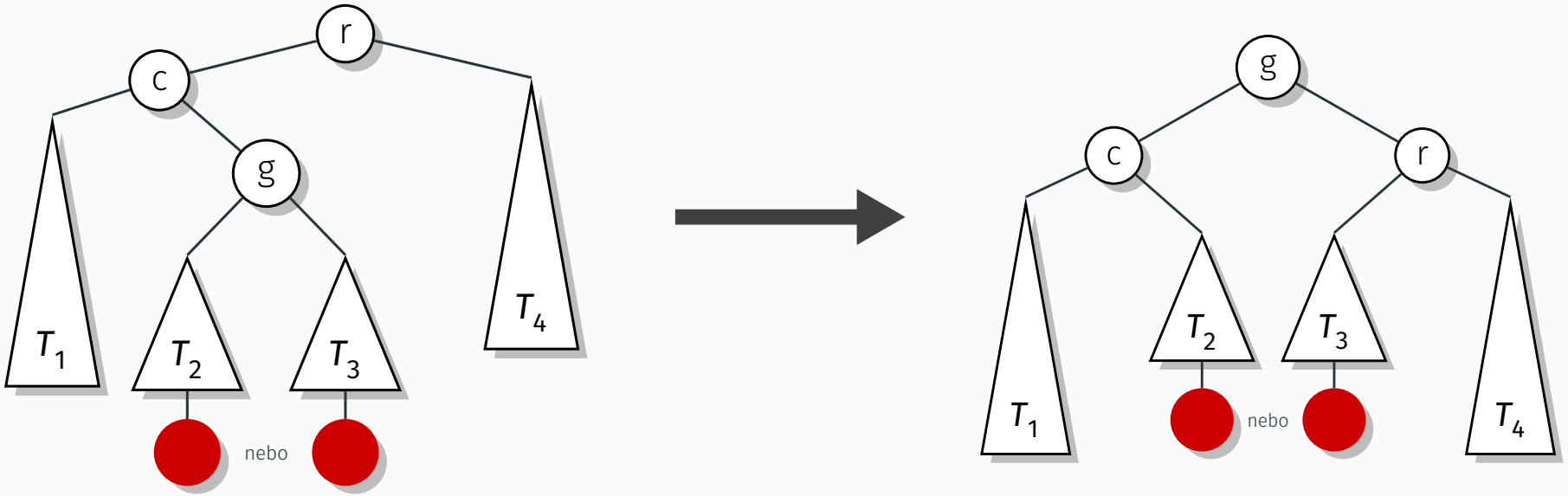
RL-rotace



AVL stromy – obecné schéma R-rotace



AVL stromy – obecné schéma LR-rotace

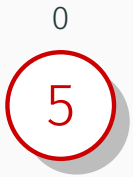


AVL stromy – vlastnosti rotací

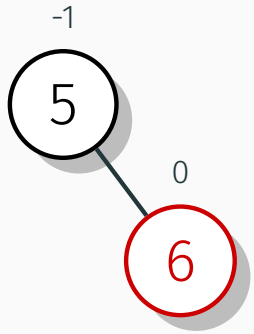
- Konstantní časová složitost – přesunují se pouze ukazatele mezi uzly, ne data.
- Rotace zachovávají uspořádání klíčů ve stromu – po dokončení rotace jsou „vlevo“ vždy menší klíče, „vpravo“ vždy větší.

AVL stromy – postupná konstrukce stromu

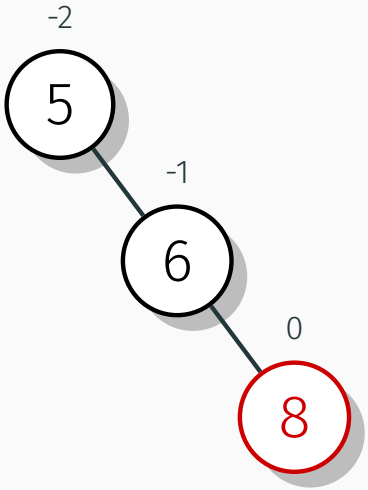
Vložení 5

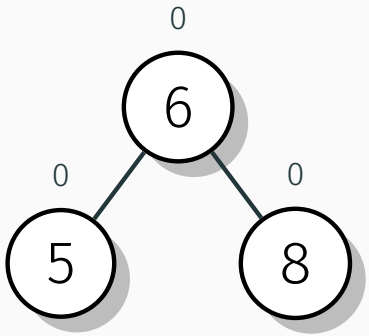


Vložení 6

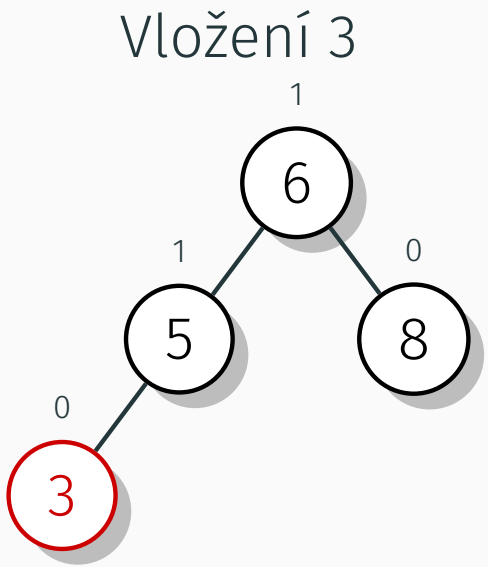


Vložení 8





AVL stromy – postupná konstrukce stromu (pokrač.)



AVL stromy – postupná konstrukce stromu (pokrač.)

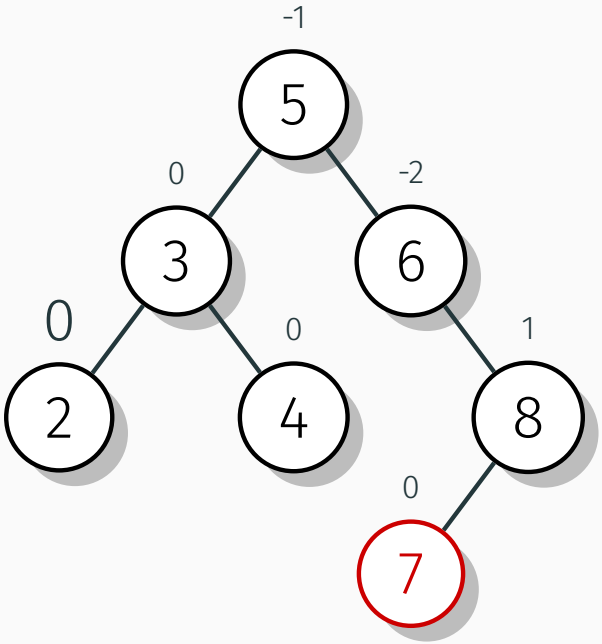
Vložení 2

AVL stromy – postupná konstrukce stromu (pokrač.)

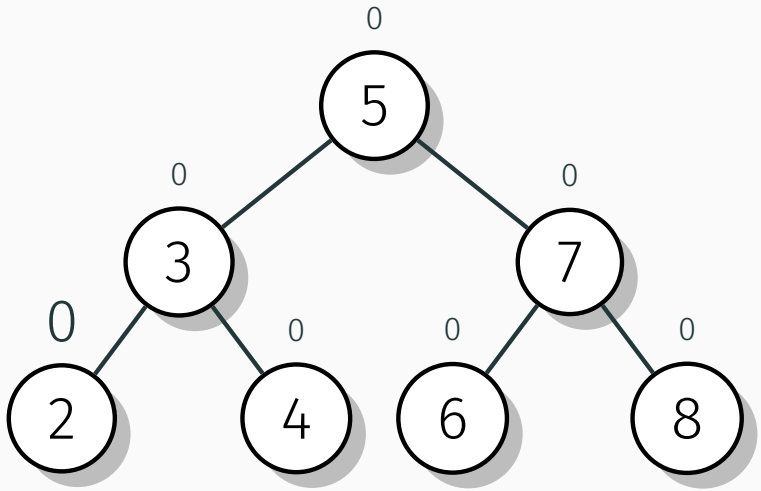
Vložení 4

AVL stromy – postupná konstrukce stromu (pokrač.)

Vložení 7



RL(6)



AVL stromy – vlastnosti

- Výška AVL stromu s n uzly je ohraničena

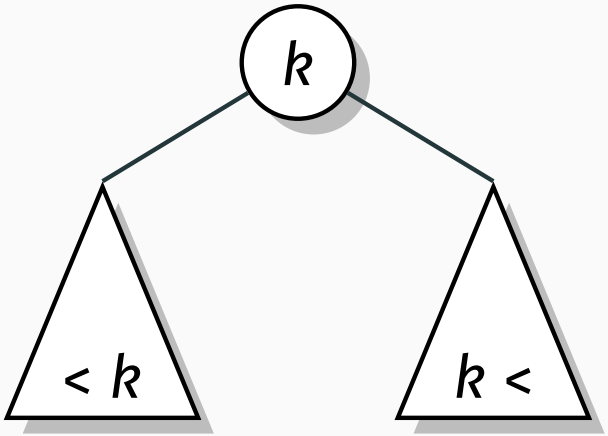
$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277$$

- Operace vyhledávání a vkládání tedy probíhají se složitostí $\Theta(\log_2 n)$ i v nejhorším případě.
- Průměrná výška AVL stromu sestrojeného z náhodné posloupnosti n klíčů je $1.01 \log_2 n + 0.1$.
- Smazání uzlu je komplikovanější, ale stále spadá do logaritmické třídy složitosti.
- Nevýhody – velké množství rotací při vyvažování stromu.

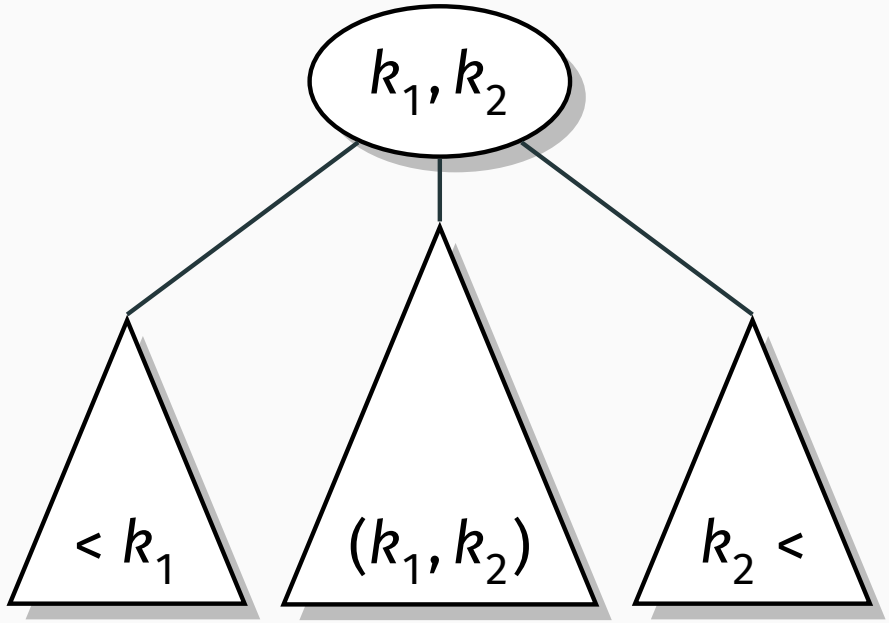
2-3 stromy

2-3 stromy – druhy uzlů

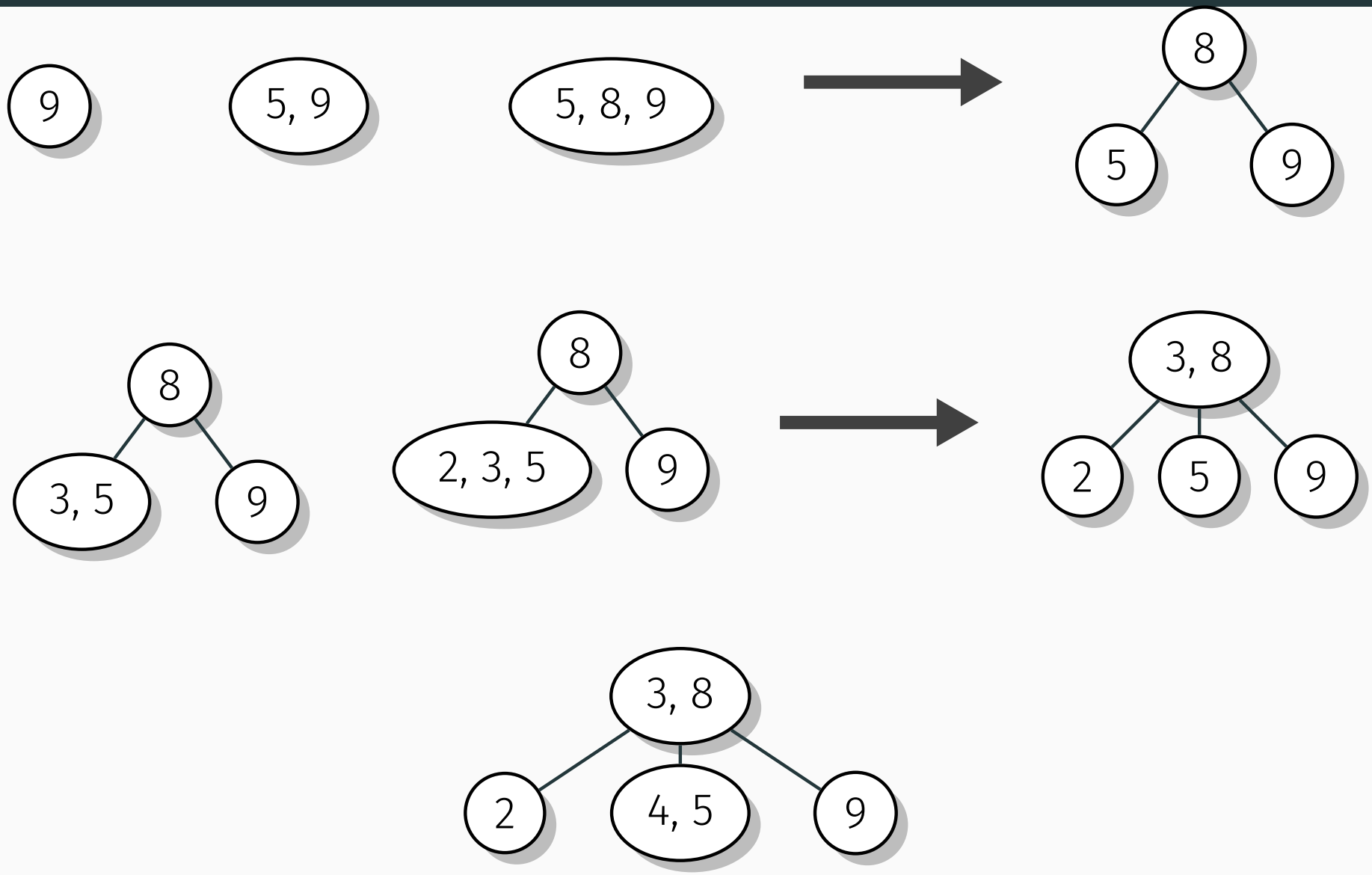
2 – uzel



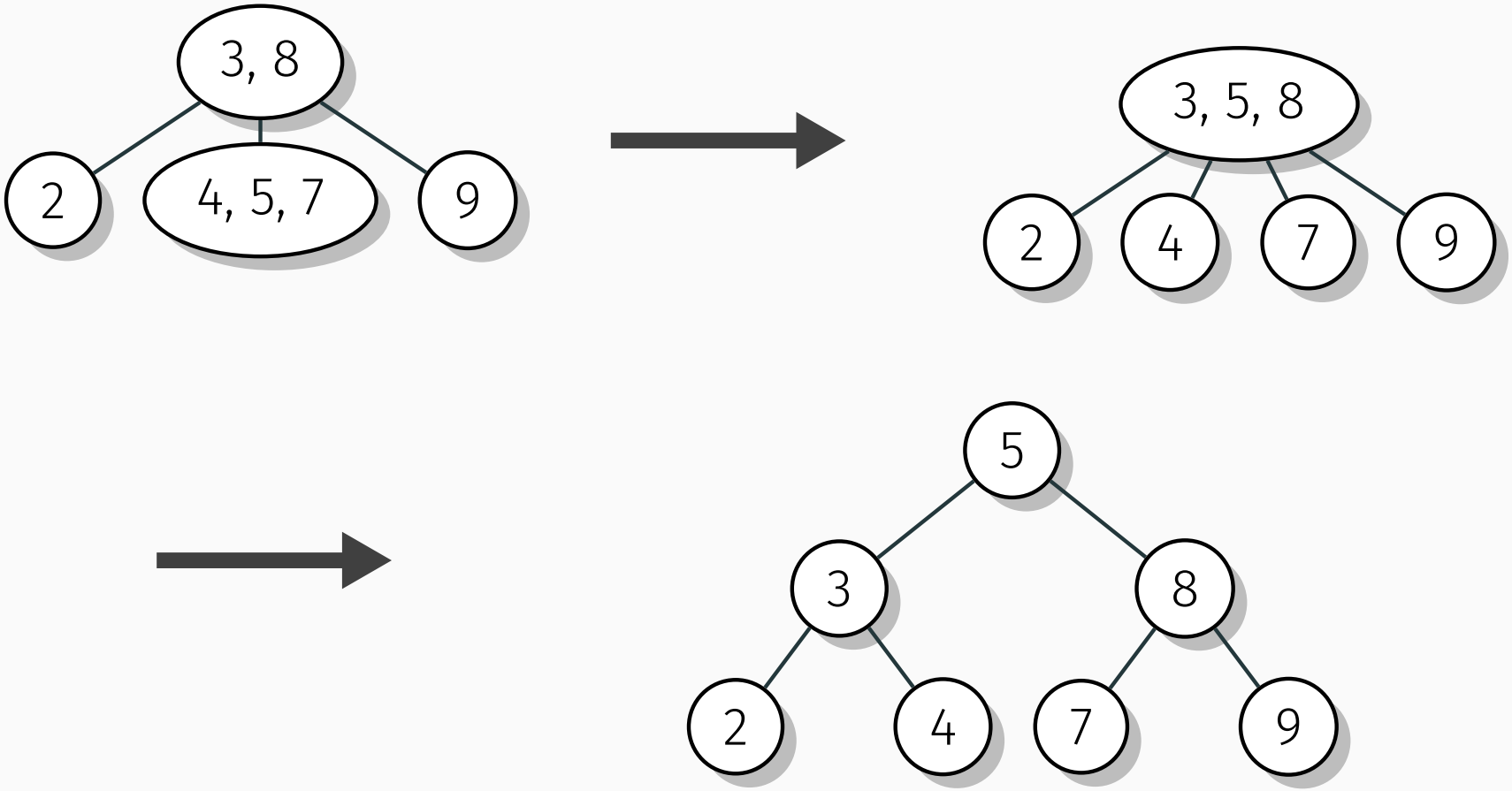
3 – uzel



Konstrukce 2-3 stromu z posloupnosti 9, 5, 8, 3, 2, 4, 7



Konstrukce 2-3 stromu z posloupnosti 9, 5, 8, 3, 2, 4, 7 (pokrač.)



Zdroje pro samostatné studium

- Kniha [2], kapitola 6.3, strany 218 – 225
- Kniha [7], kapitoly 4.4.6, 4.4.7 a 4.4.8, strany 296 – 310

Strategie řešení transformuj a vyřeš

Halda a třídění haldou

Halda (Heap)

Halda – částečně setříděná datová struktura, zvláště vhodná pro implementaci prioritní fronty.

Prioritní fronta – datová struktura chápaná jako multimnožina, kde prvky jsou řazeny podle **priority** a podporující operace:

- nalezení prvku s nejvyšší prioritou,
- odebrání prvku s nejvyšší prioritou a
- vložení nového prvku do fronty.

Využití prioritní fronty :

- plánování úloh v OS
- grafových algoritmech např. Primův, Dijkstrův atd.
- třídění haldou – **HeapSort**
- a dalších...

Halda – rozlišení terminologie

Pojem **halda** se v informatice používá pro označení:

- datové struktury a
- části operační paměti za běhu programu.

V dalším výkladu se budeme zabývat haldou výhradně jako **datovou strukturou**.

Halda

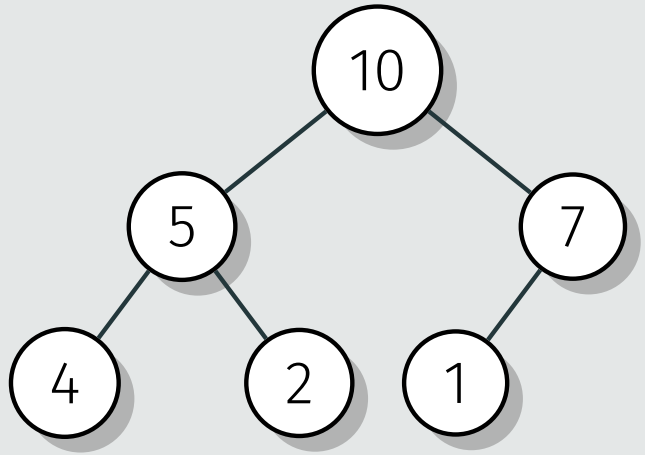
Definice

Haldu definujeme jako binární strom s jedním klíčem v každém uzlu, který splňuje následující dvě vlastnosti:

1. **kompletnost**, tj. všechna patra stromu jsou zaplněna, s výjimkou posledního. V posledním patře může zprava chybět několik listů a
2. **rodičovská dominance**, tj. klíč v každém uzlu je vždy větší nebo roven než klíče ve všech jeho potomcích. V listech je libovolný klíč vždy brán jako větší než klíče v neexistujících potomcích.

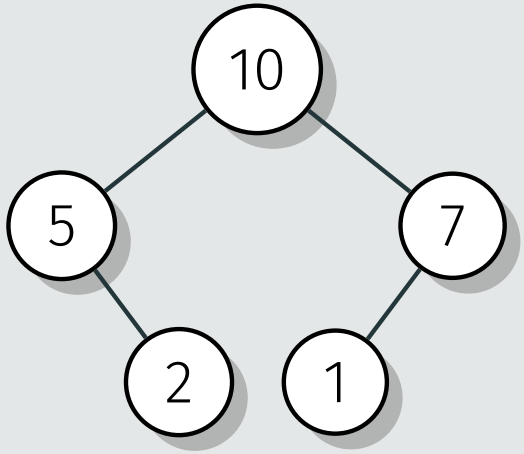
Halda – příklad

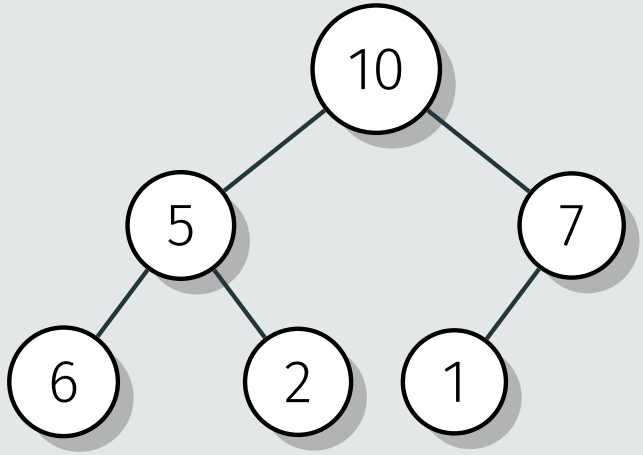
Halda



Ne každý binární strom je haldou!

Toto nejsou haldy – proč?





Halda – další vlastnosti

Pro všechny haldy lze dokázat, že:

1. Klíče na každé cestě z kořene do listu tvoří **nerostoucí** posloupnost. Jinak mezi klíči nejsou žádné vztahy, např. menší klíče v levém podstromu než v pravém atd.
2. Pro n klíčů existuje pouze jeden úplný binární strom. Jeho výška je $\lfloor \log_2 n \rfloor$.
3. Největší klíč je vždy v kořeni haldy.
4. Každý uzel v haldě je vždy kořenem haldy tvořené tímto uzlem a jeho potomky.

Halda – reprezentace v poli

V poli haldu ukládáme směrem od kořene k listům a zleva doprava: Potom:

1. vnitřní uzly – prvních $\left\lfloor \frac{n}{2} \right\rfloor$, listy zbývajících $\left\lceil \frac{n}{2} \right\rceil$,
2. potomci uzlu na pozici i, kde $1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor$, se nachází na pozicích $2i$ a $2i + 1$. A naopak rodič uzlu na pozici j, pro $2 \leq j \leq n$, se nachází na pozici $\left\lfloor \frac{j}{2} \right\rfloor$.

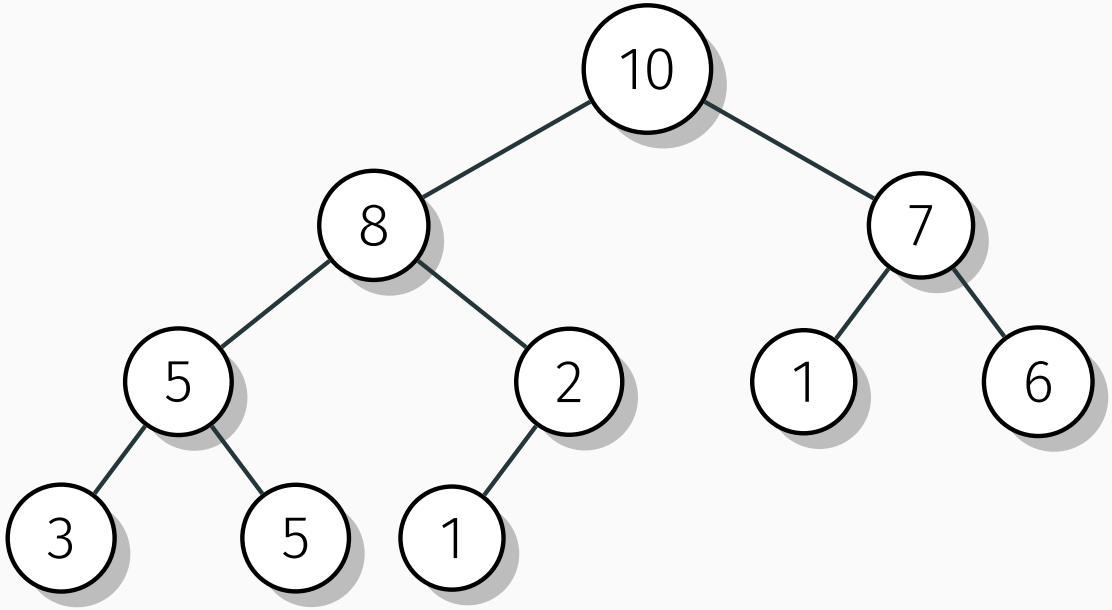
Poznámka

Haldu lze definovat jako pole $H[1 \ldots n]$ ve kterém pro každý prvek na indexu i platí

$$H[i] \geq max\{H[2i], H[2i + 1]\}$$

pro všechna $i = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$.

Halda – reprezentace v poli, příklad



index	1	2	3	4	5	6	7	8	9	10
klíč	10	8	7	5	2	1	6	3	5	1
	vnitřní uzly					listy				

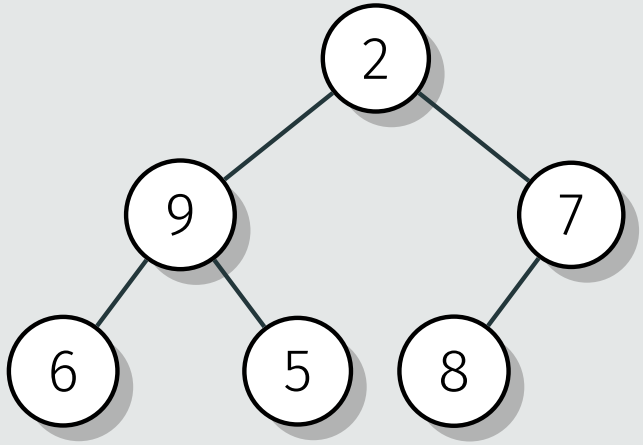
Konstrukce haldy

Haldu lze kostruovat dvěma způsoby:

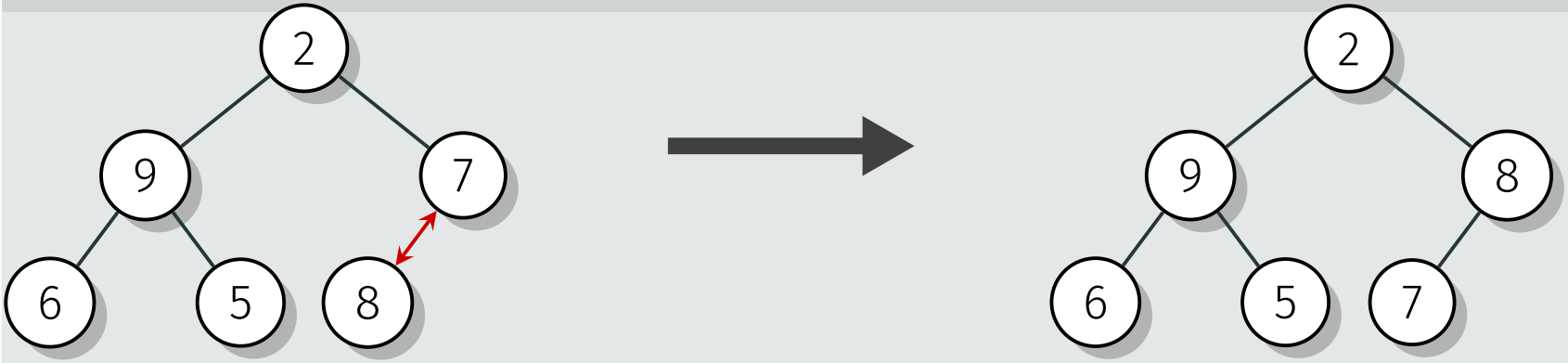
1. **zdola nahoru** a
2. shora dolů.

Konstrukce haldy zdola nahoru – příklad

Výchozí stav haldy

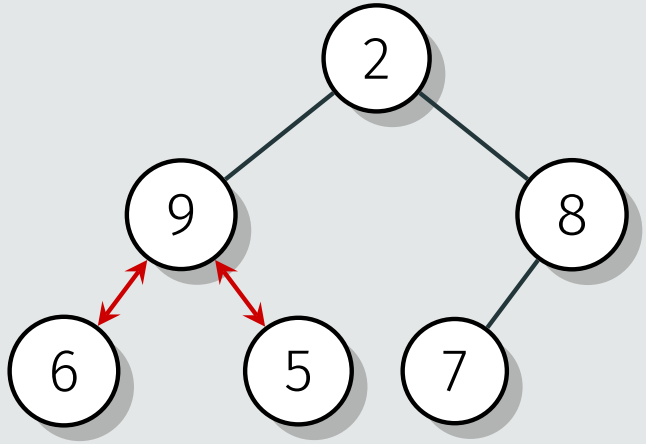


Krok 1

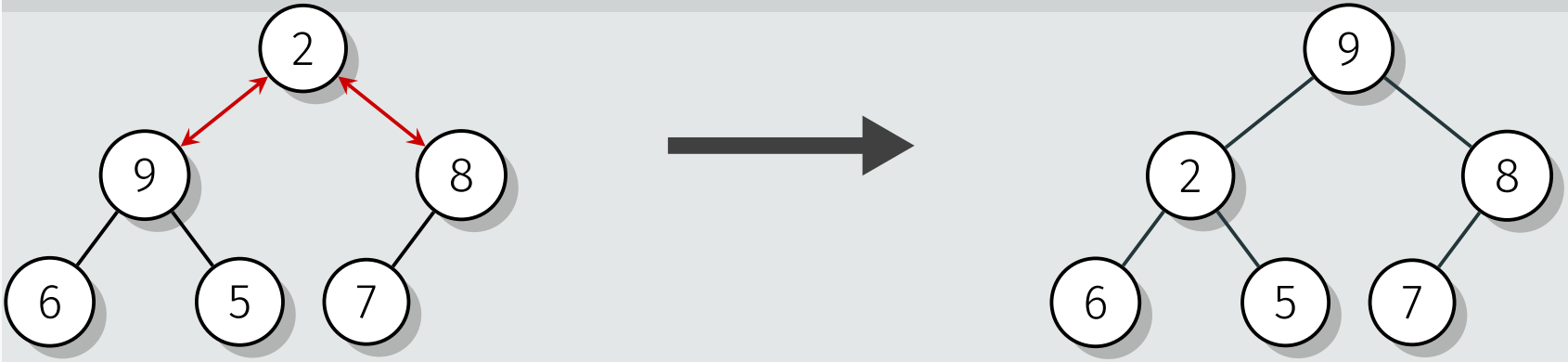


Konstrukce haldy zdola nahoru – příklad (pokrač.)

Krok 2

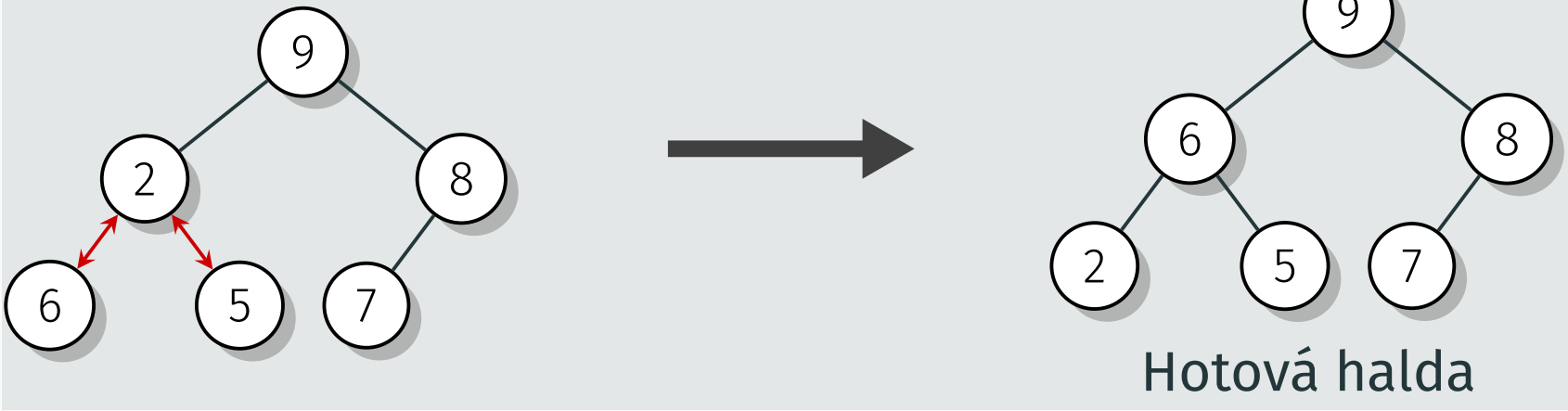


Krok 3a



Konstrukce haldy zdola nahoru – příklad (pokrač.)

Krok 3b



Konstrukce haldy zdola nahoru

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole, i kořen budované haldy

Výstup: Halda s kořenem na indexu i

```
1  procedure Heapify(A, n, i)
2      largest ← i;
3      l ← 2 * i + 1;
4      r ← 2 * i + 2;
5      if l < n ∧ A[l] > A[largest] then largest ← l ;
6      if r < n ∧ A[r] > A[largest] then largest ← r ;
7      if largest ≠ i then
8          Swap (A[i], A[largest]);
9          Heapify (A, n, largest);
10     end
11 end
```

Konstrukce haldy zdola nahoru

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Halda v poli A

```
1 procedure MakeHeap(A, n)
2 |   for i ← ⌊n/2⌋ - 1 downto 0 do
3 |   |   Heapify (A, n, i);
4 |   end
5 end
```

Konstrukce haldy zdola nahoru – časová složitost

Pro jednoduchost předpokládejme, že $n = 2^k - 1$, tj. halda tvoří úplný binární strom.

Výška haldy je pak $h = \lfloor \log_2 n \rfloor$, což lze psát jako

$$\begin{aligned} \lceil \log_2(n+1) \rceil - 1 &= \lceil \log_2(2^k - 1 + 1) \rceil - 1 \\ &= \lceil \log_2(2^k) \rceil - 1 \\ &= k - 1 \end{aligned}$$

Konstrukce haldy zdola nahoru – časová složitost (pokrač.)

Poznámka

Výraz $\lceil \log_2(n+1) \rceil$ lze interpretovat jako „výška haldy s $n+1$ prvky“. Předpokládali jsme úplný binární strom $\Rightarrow$ strom s $n+1$ prvky má určitě o jednu úroveň více než strom s n prvky.

Každý klíč z úrovně i se bude při konstrukci haldy, v nejhorší případě, posunovat až do listu, tj. na úroveň h.

Posun o jednu úroveň vyžaduje dvě porovnání:

1. nalezení většího z obou potomků a
2. test, zda je nutná výměna s rodičem.

Konstrukce haldy zdola nahoru – časová složitost (pokrač.)

Počet porovnání je tedy $2(h - i)$.

Celkový počet porovnání bude v nejhorším případě roven

$$\begin{aligned} C(n) &= \sum_{i=0}^{h-1} \sum_{\text{klíče úrovně } i} 2(h - i) \\ &= \sum_{i=0}^{h-1} 2(h - i)2^i = 2h \sum_{i=0}^{h-1} 2^i - 2 \sum_{i=0}^{h-1} i2^i \\ &= 2n - 2 \log_2(n + 1)) \end{aligned}$$

Konstrukce haldy s n prvky vyžaduje, v nejhorším případě, méně než $2n$ porovnání.

Konstrukce haldy zdola nahoru – časová složitost (pokrač.)

Poznámka

V odvození jsme použili vzorce:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

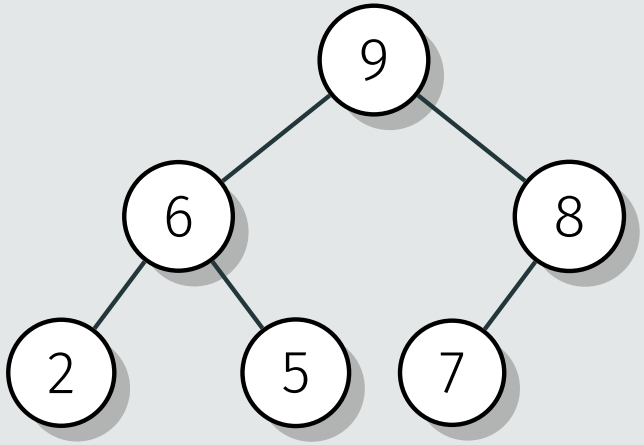
$$\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

Konstrukce haldy shora dolů

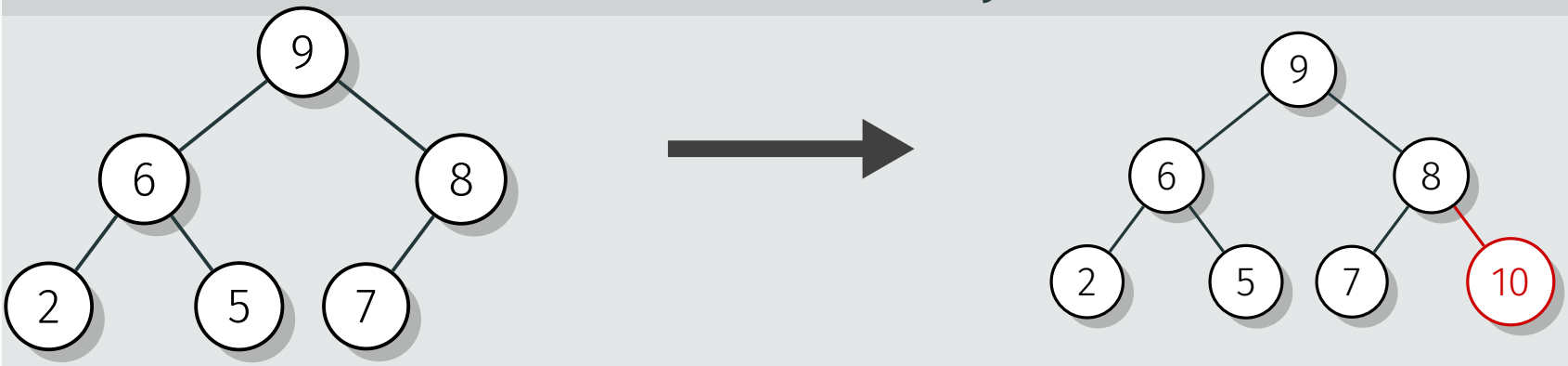
- Opakované vkládání nového klíče do již existující haldy.
  1. Nový klíč vložíme na konec haldy.
  2. Nový klíč porovnáme s rodičem a případně nový klíč přesuneme o patro výš.
  3. Takto postupujeme dokud nenarazíme na většího rodiče nebo dojdeme do kořene haldy.
- Výška haldy s n prvky je $\approx \log_2 n$, tudíž složitost vložení klíče do haldy je $O(\log n)$.
- Konstrukce shora dolů je tedy složitější než zdola nahoru.

Konstrukce haldy shora dolů – příklad

Výchozí stav haldy

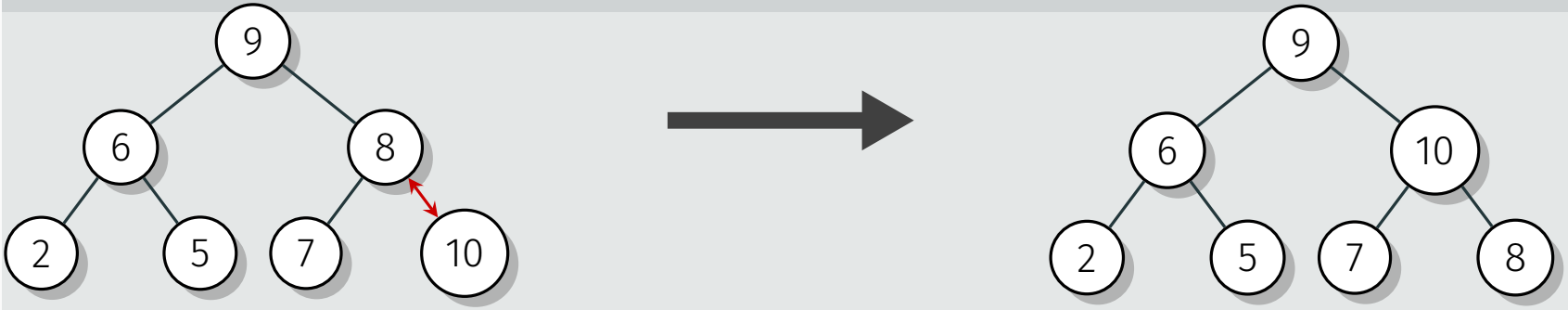


Krok 1 – vložení klíče 10 na konec haldy

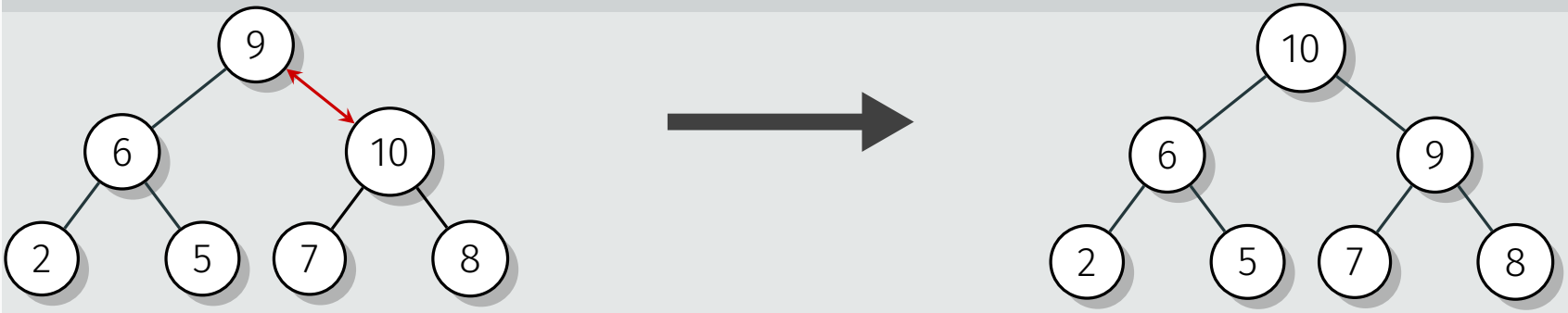


Konstrukce haldy shora dolů – příklad (pokrač.)

Krok 2a – porovnání klíče 10 s rodičem



Krok 2b – porovnání klíče 10 s rodičem



Odstranění největšího klíče z haldy

Princip algoritmu:

1. Výměna klíče v kořeni s klíčem na konci haldy.
2. Zmenšení haldy o jedna.
3. Obnova haldy – otestovat, zda je klíč v rodiči větší než klíče v obou potomcích a případně provést výměnu. Postup opakovat dokud nebude rodičovský klíč větší než klíče v potomcích.

Poznámka

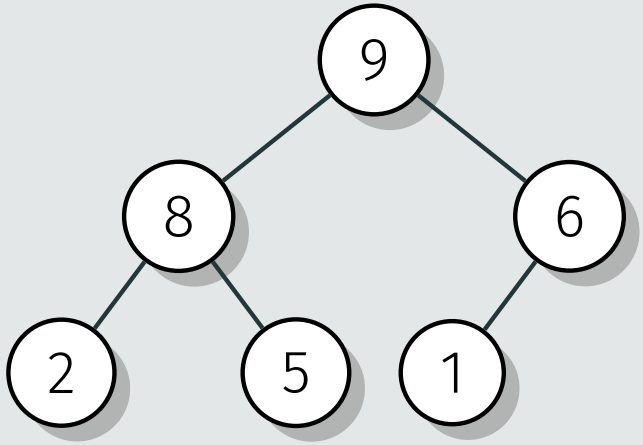
Principiálně, lze z haldy odebrat jakýkoliv klíč. Ale tato operace nemá praktický význam.

Odstranění největšího klíče z haldy – složitost algoritmu

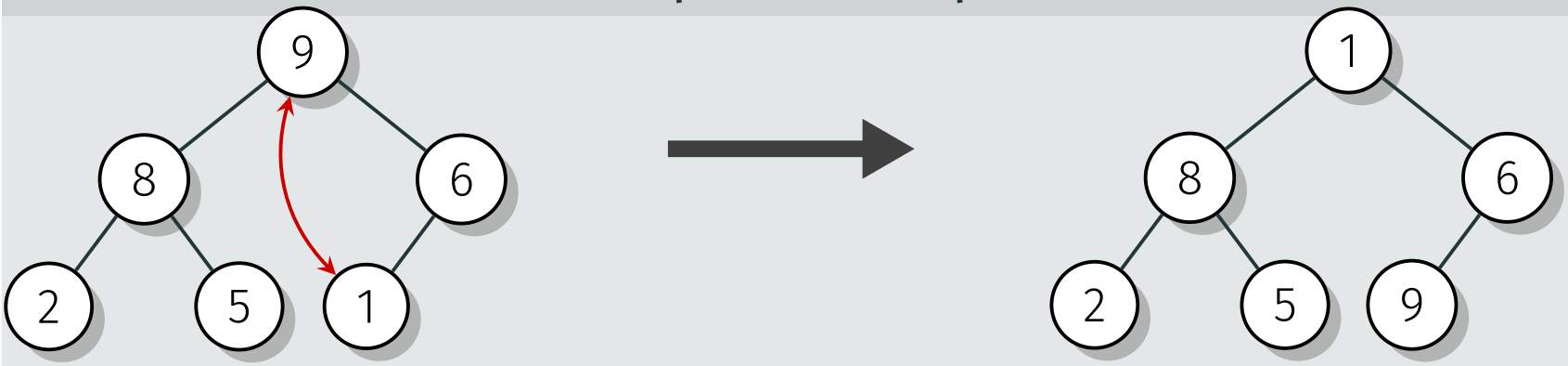
- Počet porovnání nutných pro obnovení haldy je úměrný výšce haldy – „posunujeme“ klíč z kořene po patrech dolů.
- Porovnáváme vždy rodiče s oběma potomky – musíme najít největšího z dané trojice.
- Výška haldy je $h \approx \log_2 n$, počet porovnání nebude tedy větší než $2h$.
- Složitost algoritmu je tedy $O(\log n)$.

Odstranění největšího klíče z haldy – příklad

Výchozí stav haldy

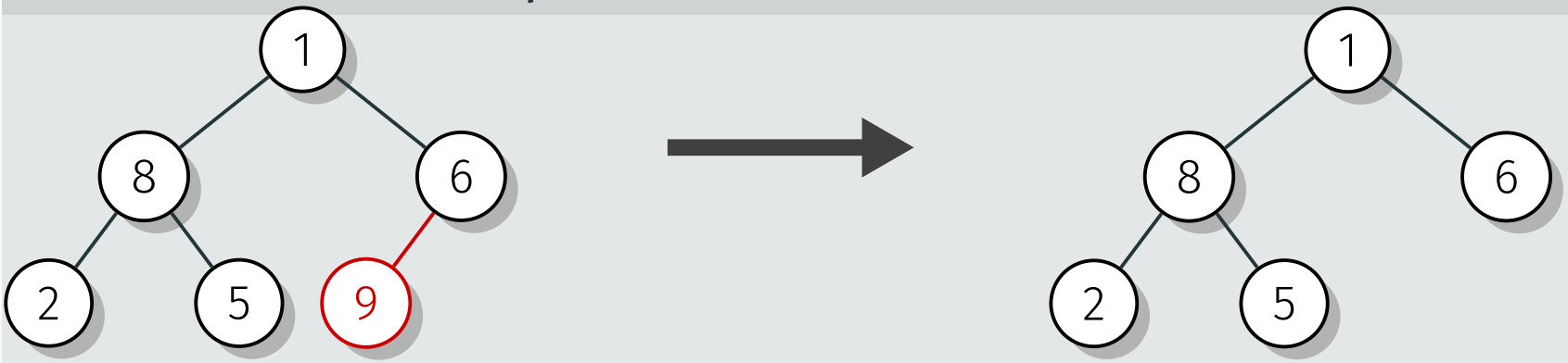


Krok 1 – záměna kořene s posledním prvkem

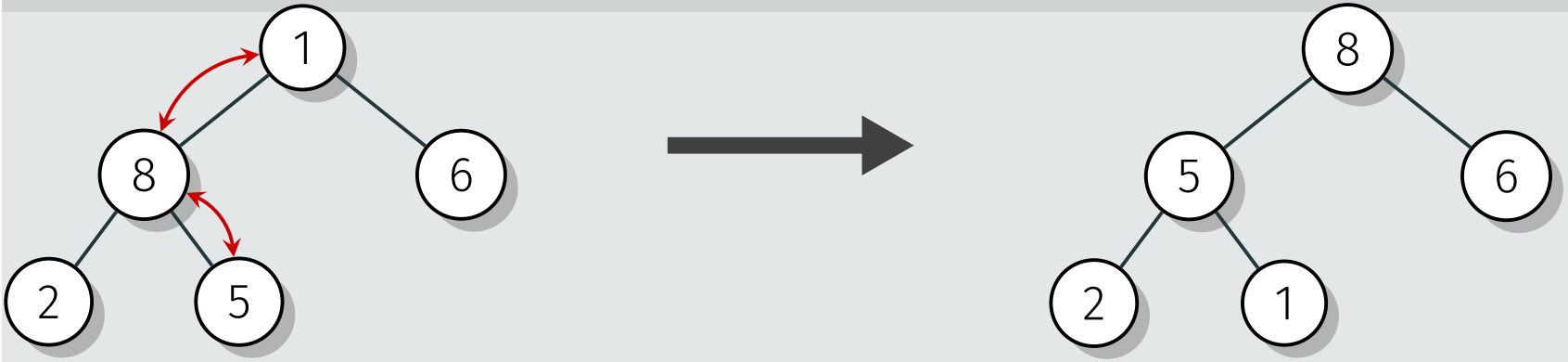


Odstranění největšího klíče z haldy – příklad (pokrač.)

Krok 2 – odstranění posledního uzlu



Krok 3 – obnovení haldy



Třídění haldou – HeapSort

Algoritmus pracuje ve dvou fázích:

Konstrukce haldy : pro dané pole je sestavena halda.

Odstranění maxima : $(n - 1)$-krát je aplikován algoritmus pro odstranění největšího klíče z, postupně se zmenšující, haldy.

Třídění haldou – HeapSort

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole
Výstup: Setříděné pole A

```
1 procedure HeapSort(A, n)
2 |   MakeHeap (A, n);
3 |   for i ← n − 1 downto 0 do
4 |   |   Swap (A[0], A[i]);
5 |   |   Heapify (A, i, 0);
6 |   end
7 end
```

Třídění haldou – složitost algoritmu

- Složitost první fáze je $O(n)$.
- Ve druhé fázi postupně odstraňujeme největší klíč z haldy o klesající velikosti $n, n-1, \ldots, 2$. Počet porovnání $C(n)$ je

$$\begin{aligned} C(n) &\leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \\ &\leq 2\sum_{i=1}^{n-1} \log_2 i \\ &\leq 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \leq 2n \log_2 n \end{aligned}$$

Platí tedy $C(n) \in O(n \log n)$.

Třídění haldou – složitost algoritmu (pokrač.)

- Pro obě fáze dostáváme $O(n) + O(n \log n) = O(n \log n)$.
- Další analýzou složitosti lze dokázat, že stejná složitost platí i pro průměrný případ. Tedy $\Theta(n \log n)$.
- Třídění haldou je srovnatelné s tříděním sléváním.
- V praxi je však pomalejší než QuickSort.

Zdroje pro samostatné studium

- Kniha [2], kapitola 6.4, strany 226 – 232
- Kniha [3], kapitoly 6.1 až 6.4, strany 161 – 172

Strategie řešení transformuj a vyřeš

Hornerovo schéma

Hodnota polynomu v bodě

Zadání

Máme dán polynom

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Naším úkolem je vypočítat hodnotu polynomu $p(x)$ v bodě x_0.

Motivace

- Polynomy se využívají pro aproximaci funkcí aneb
  1. Jak procesor vypočte hodnotu funkce $\sin(x)$?
  2. Kde se vzaly hodnoty funkce $\sin(x)$ v matematických tabulkách?

  Pomocí Taylorova rozvoje funkce, což je polynom!
- Rychlá Fourierova trasformace

Taylorův rozvoj funkce $y = f(x)$

Funkci $f(x)$, která má v bodě a konečné derivace do řádu $n+1$ lze, v okolí bodu a, psát jako rozvoj

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_{n+1}^{f,a}(x)$$

Pro $a = 0$ se rozvoj nazývá Maclaurinův

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + R_{n+1}^{f,0}(x)$$

Taylorův rozvoj funkce $y = \sin(x)$ v bodě 0

$$\sin(x) = \sin(0) + \frac{\sin'(0)}{1!}x + \frac{\sin''(0)}{2!}x^2 + \cdots + \frac{\sin^{(n)}(0)}{n!}x^n + R_{n+1}^{\sin,0}(x)$$

Derivace

$$\sin^{(1)} 0 = \cos 0 = 1 \qquad \sin^{(2)} 0 = -\sin 0 = 0$$

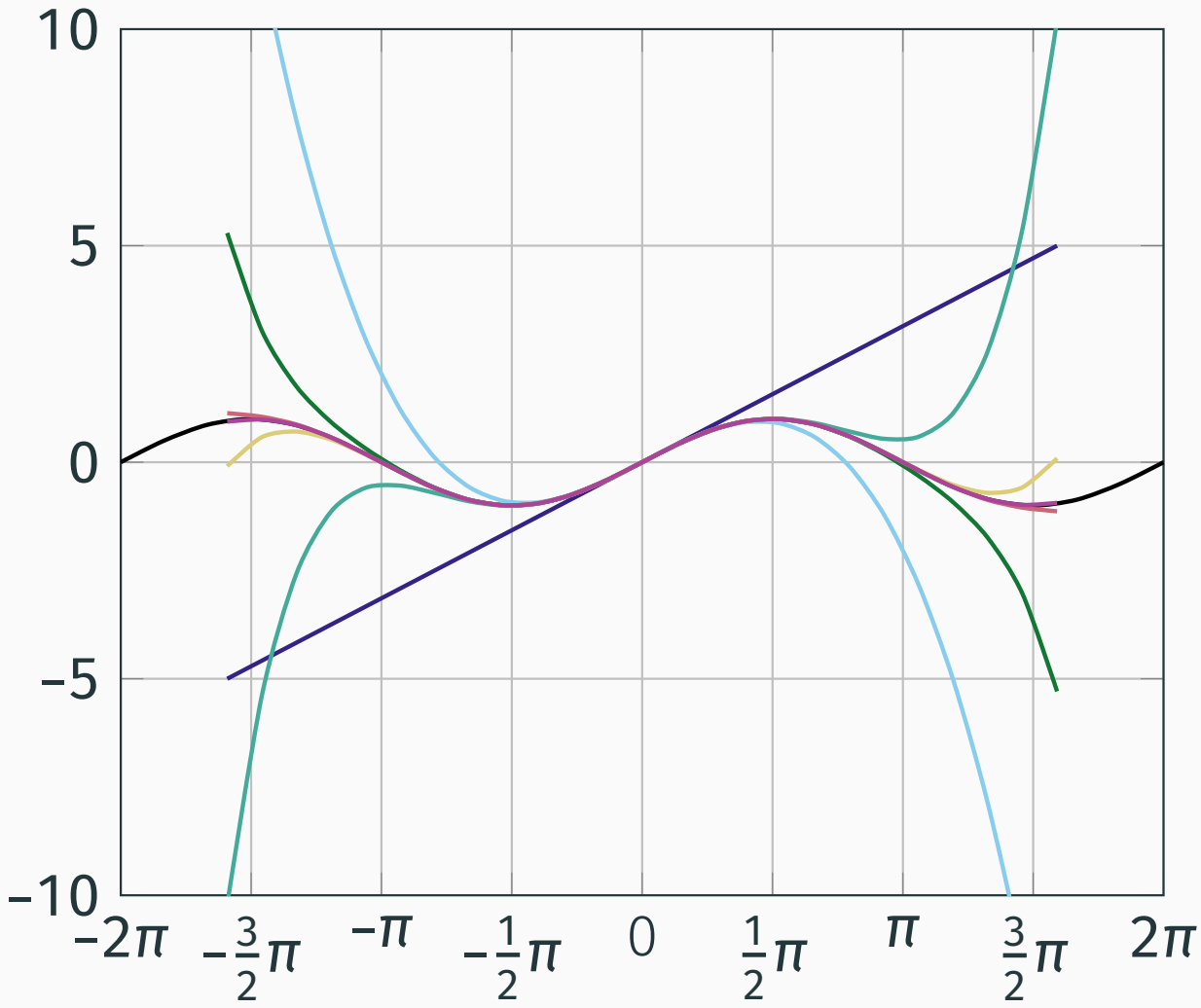
$$\sin^{(3)} 0 = -\cos 0 = -1 \qquad \sin^{(4)} 0 = \sin 0 = 0$$

$$\sin(x) = 0 + \frac{1}{1!}x + \frac{0}{2!}x^2 + \frac{-1}{3!}x^3 + \frac{0}{4!}x^4 + \cdots + R_{n+1}^{\sin,0}(x)$$

Aproximace polynomem 13-tého stupně

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!}$$

Taylorův rozvoj funkce $y = \sin(x)$ v bodě 0

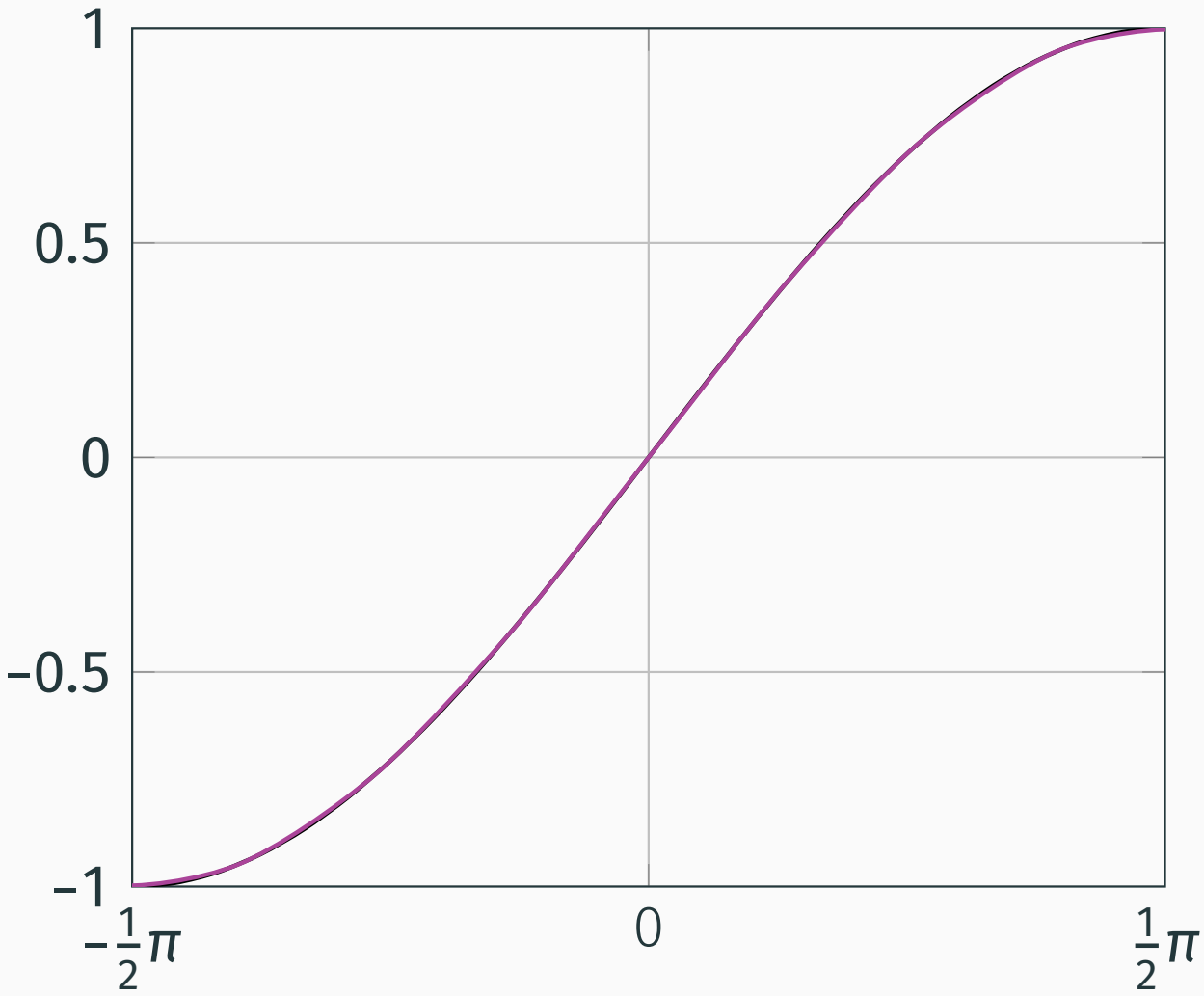


Taylorův rozvoj:

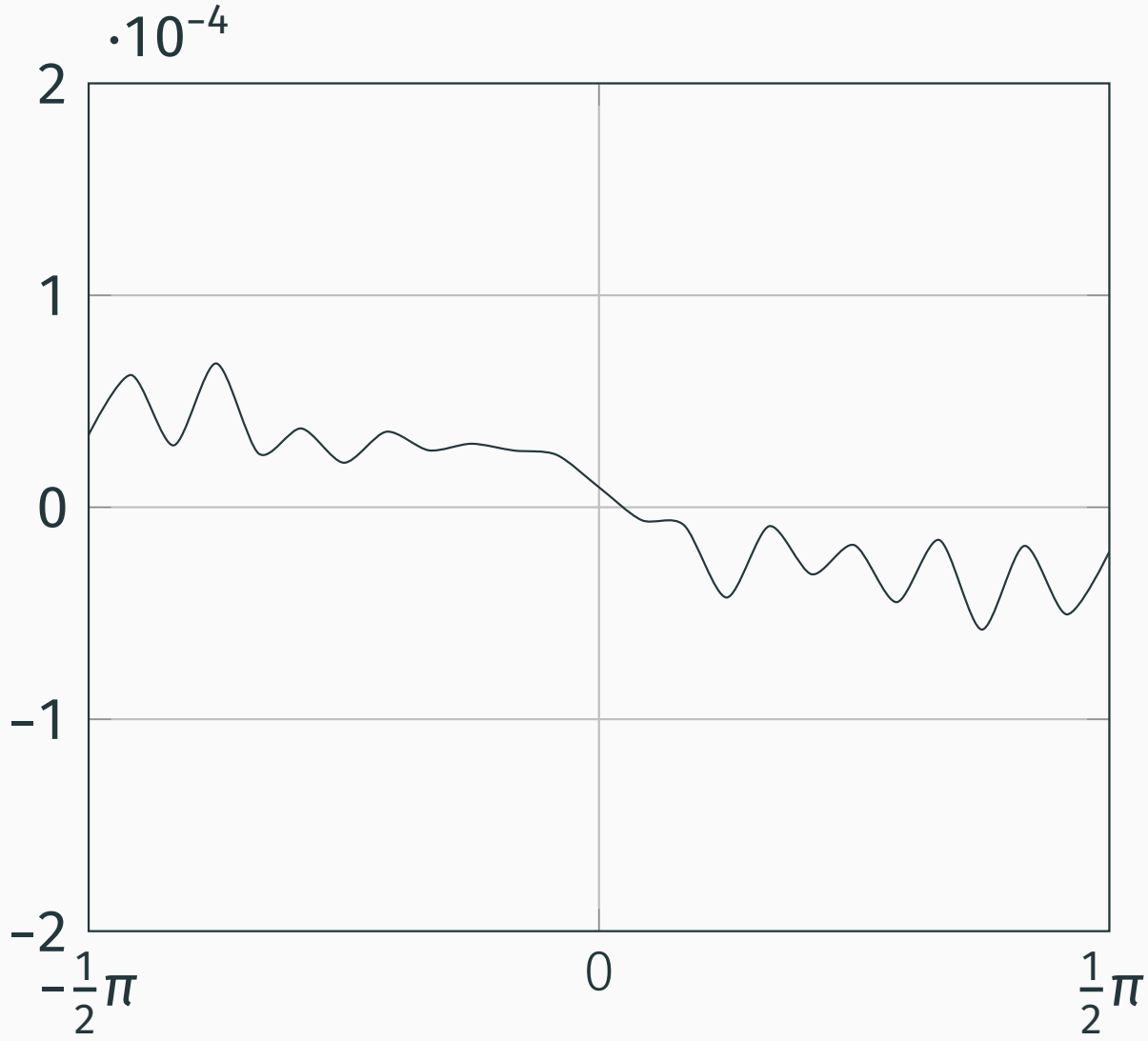
- stupně 1
- stupně 3
- stupně 5
- stupně 7
- stupně 9
- stupně 11
- stupně 13

Funkce $y = \sin(x)$ je zobrazena černě.

Taylorův rozvoj funkce $y = \sin(x)$ stupně 13 v bodě 0

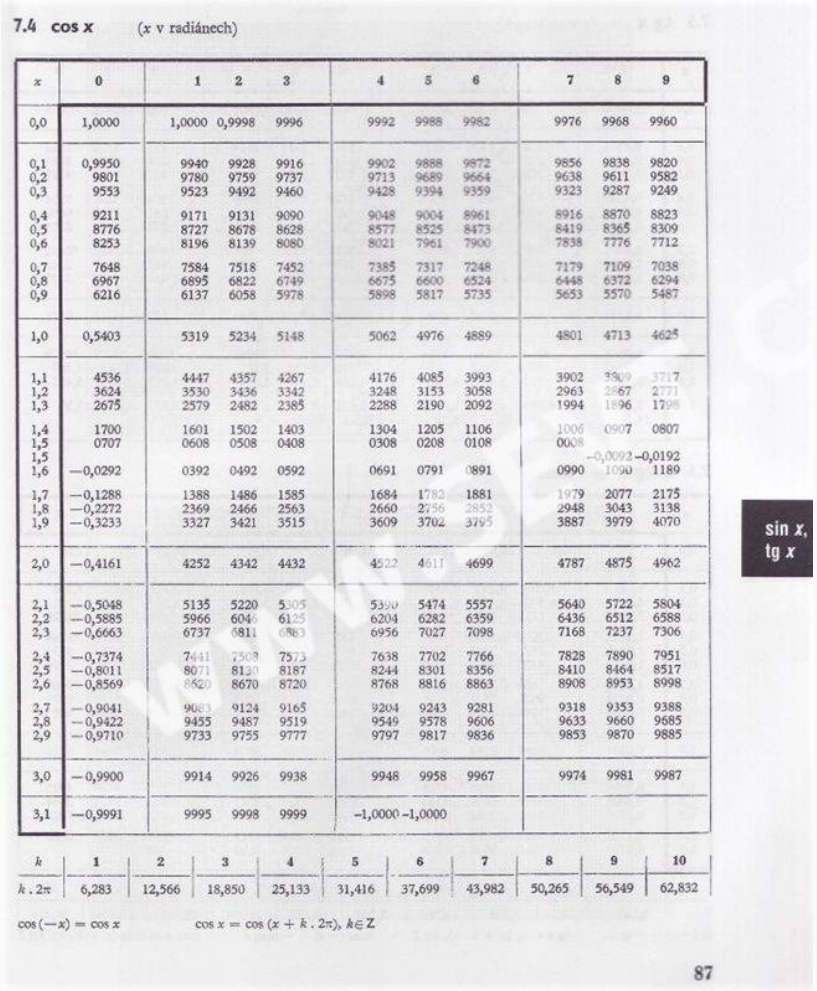


Taylorův rozvoj funkce $y = \sin(x)$ v bodě 0, chyba aproximace



Tabulky hodnot funkcí

- Taylorovým rozvojem lze aproximovat hodnotu požadované funkce a sestavit tabulky.
- Ruční výpočet – náročné a zatíženo obrovským množstvím chyb.
- Průlomová myšlenka – k numerickým výpočtům není nutná inteligence! Lze je provádět **strojově**!



7.4 cos x (x v radiánech)

x	0	1	2	3	4	5	6	7	8	9
0,0	1,0000	1,0000	0,9998	9996	9992	9988	9982	9976	9968	9960
0,1	0,9950	9940	9928	9916	9902	9888	9872	9856	9838	9820
0,2	9801	9780	9759	9737	9713	9689	9664	9638	9611	9582
0,3	9553	9523	9492	9460	9428	9394	9359	9323	9287	9249
0,4	9211	9171	9131	9090	9048	9004	8961	8916	8870	8823
0,5	8776	8727	8678	8628	8577	8525	8473	8419	8365	8309
0,6	8253	8196	8139	8080	8021	7961	7900	7838	7776	7712
0,7	7648	7584	7518	7452	7385	7317	7248	7179	7109	7038
0,8	6967	6895	6822	6749	6675	6600	6524	6448	6372	6294
0,9	6216	6137	6058	5978	5898	5817	5735	5653	5570	5487
1,0	0,5403	5319	5234	5148	5062	4976	4889	4801	4713	4625
1,1	4536	4447	4357	4267	4176	4085	3993	3902	3809	3717
1,2	3624	3530	3436	3342	3248	3153	3058	2963	2867	2771
1,3	2675	2579	2482	2385	2288	2190	2092	1994	1896	1798
1,4	1700	1601	1502	1403	1304	1205	1106	1006	0907	0807
1,5	0707	0608	0508	0408	0308	0208	0108	0008		
1,6	−0,0292	0392	0492	0592	0691	0791	0891	0990	−0,0092	−0,0192
1,7	−0,1288	1388	1486	1585	1684	1782	1881	1979	2077	2175
1,8	−0,2272	2369	2466	2563	2660	2756	2852	2948	3043	3138
1,9	−0,3233	3327	3421	3515	3609	3702	3795	3887	3979	4070
2,0	−0,4161	4252	4342	4432	4522	4611	4699	4787	4875	4962
2,1	−0,5048	5135	5220	5305	5390	5474	5557	5640	5722	5804
2,2	−0,5885	5966	6046	6125	6204	6282	6359	6436	6512	6588
2,3	−0,6663	6737	6811	6883	6956	7027	7098	7168	7237	7306
2,4	−0,7374	7441	7508	7573	7638	7702	7766	7828	7890	7951
2,5	−0,8011	8071	8130	8187	8244	8301	8356	8410	8464	8517
2,6	−0,8569	8620	8670	8720	8768	8816	8863	8908	8953	8998
2,7	−0,9041	9083	9124	9165	9204	9243	9281	9318	9353	9388
2,8	−0,9422	9455	9487	9519	9549	9578	9606	9633	9660	9685
2,9	−0,9710	9733	9755	9777	9797	9817	9836	9853	9870	9885
3,0	−0,9900	9914	9926	9938	9948	9958	9967	9974	9981	9987
3,1	−0,9991	9995	9998	9999	−1,0000	−1,0000				

k	1	2	3	4	5	6	7	8	9	10
k . 2π	6,283	12,566	18,850	25,133	31,416	37,699	43,982	50,265	56,549	62,832

$\cos(-x) = \cos x$ $\cos x = \cos(x + k \cdot 2\pi),\ k \in Z$

87

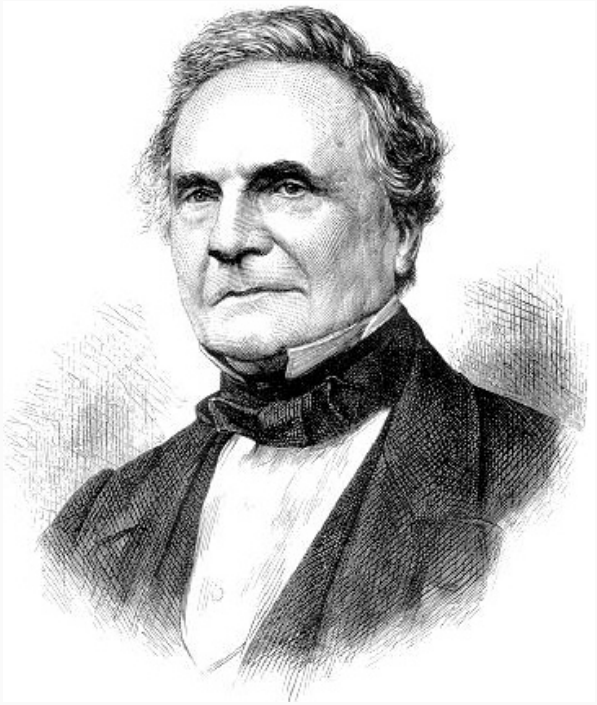
sin x, tg x

Charles Babbage – Difference Engine

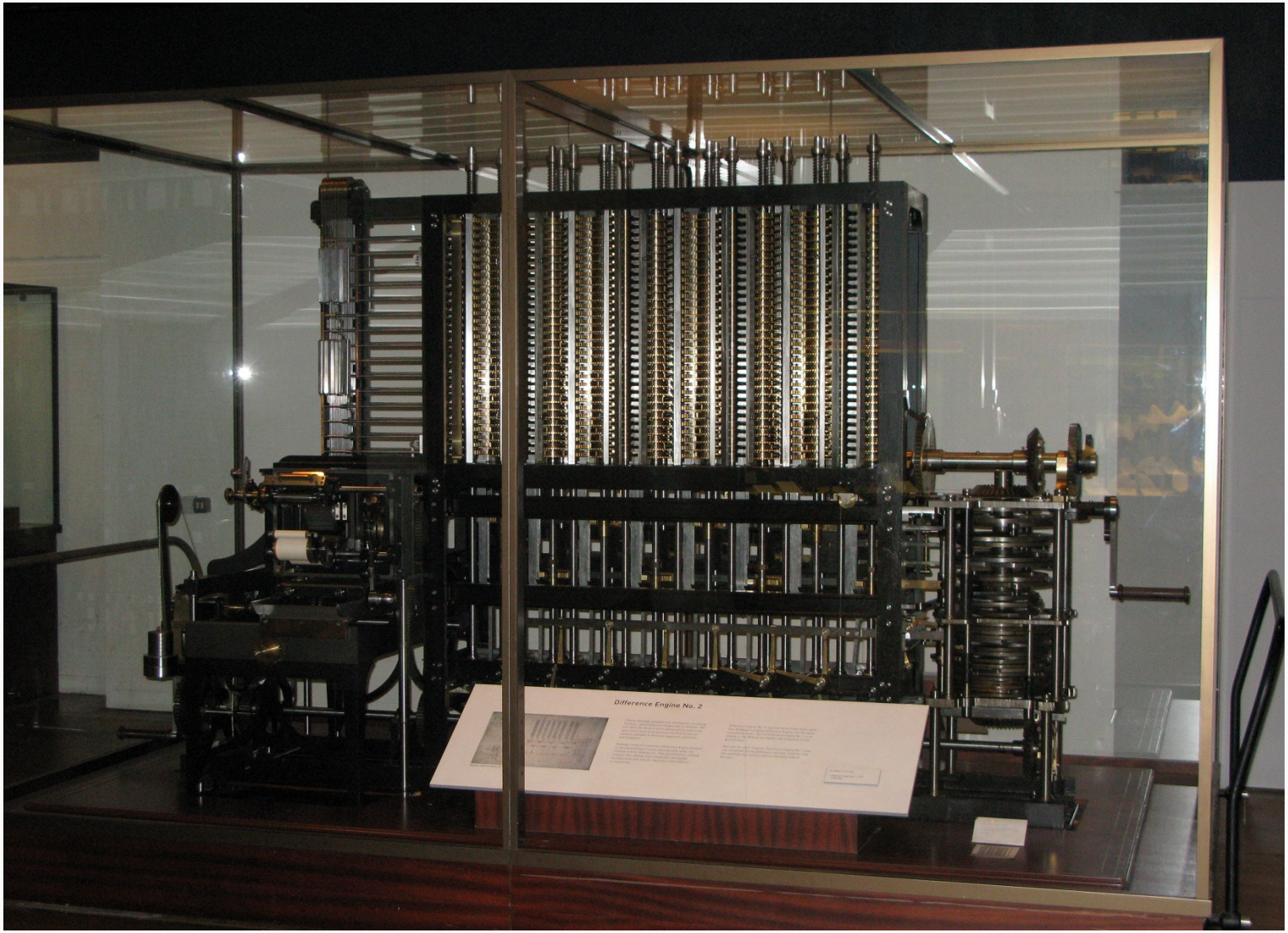
Difference Engine

- první programovatelný počítač na světě
- 1819 – zahájení prací
- 1822 – dokončen prototyp
- 1823 – zahájeny práce na velkém stroji
- 1833 – přerušení prací
- 1842 – ukončení vládní podpory, na projekt vynaloženo 17 tisíc liber, stroj nebyl nikdy dokončen
- 1991 – funkční replika!

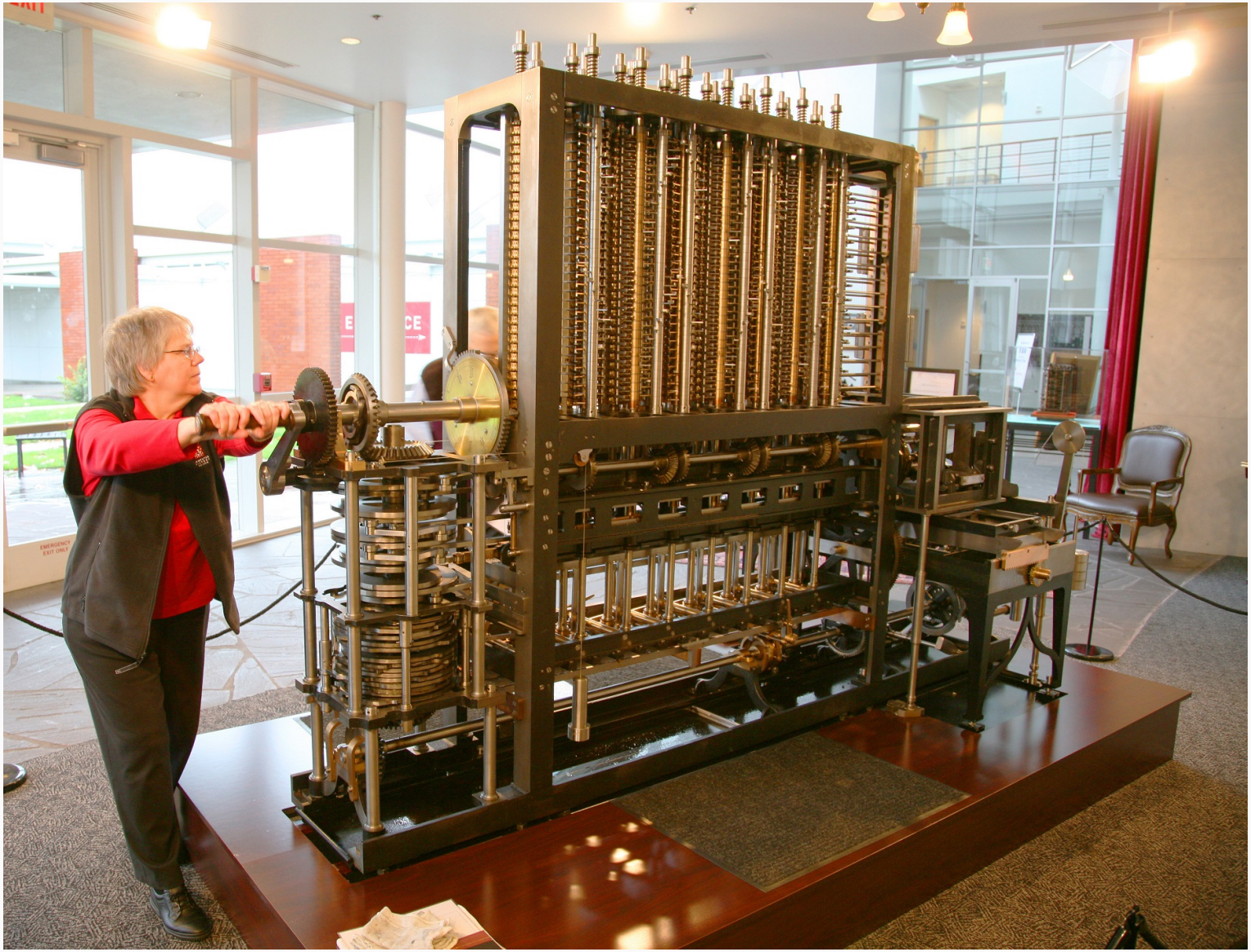
Charles Babbage
(1791 – 1871)



Difference Engine



Difference Engine



První programátor na světě?!

Augusta Ada King, hraběnka z Lovelace (1815 – 1852)

Programátorka **Analytical Engine**, (Babbage 1837), což byl první obecně použitelný turingovsky úplný počítač.

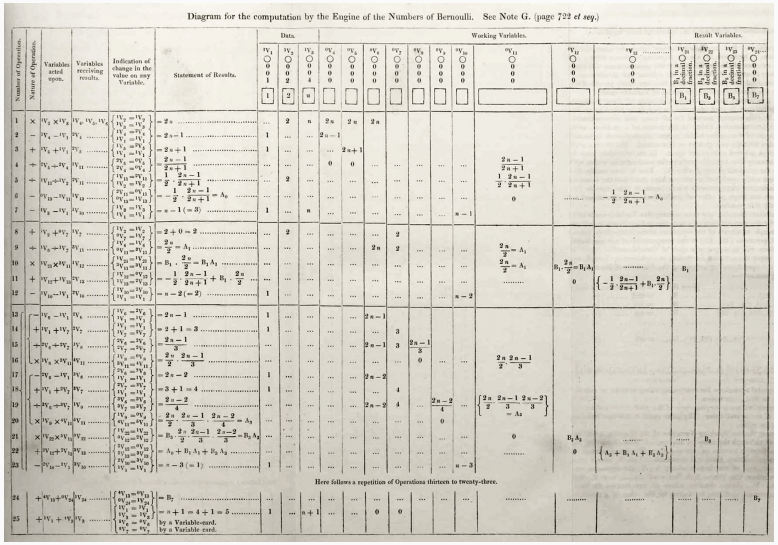


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

Hornerovo schéma – transformace

Základní myšlenka:

- transformace polynomu na jiný tvar,
- postupně vytýkáme z částí polynomu proměnnou x.

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\ &= a_0 + x\left(a_1 + a_2 x + \cdots + a_{n-1} x^{n-2} + a_n x^{n-1}\right) \\ &= a_0 + x\left(a_1 + x\left(a_2 + \cdots + a_{n-1} x^{n-3} + a_n x^{n-2}\right)\right) \\ &\vdots \\ &= a_0 + x\Big(a_1 + x\Big(a_2 + \cdots + x\big(a_{n-1} + a_n x\big)\ldots\Big)\Big) \end{aligned}$$

Že tato rovnost platí, je snadno vidět postupným roznásobením všech závorek.

Hornerovo schéma – výpočet

Hodnotu $p(x_0)$ počítáme „z vnitřku“ závorek, postupně počítáme hodnoty b_i

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= a_{n-1} + b_n x_0 \\ b_{n-2} &= a_{n-2} + b_{n-1} x_0 \\ &\vdots \\ b_0 &= a_0 + b_1 x_0 \end{aligned}$$

Hodnota b_0 je pak rovna $p(x_0)$, neboť

$$p(x_0) = a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(a_{n-1} + a_n x_0)\ldots\big)\Big)$$

Hornerovo schéma – výpočet (pokrač.)

a postupným dosazováním za b_i dostáváme

$$
\begin{aligned}
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\Big(a_2 + \cdots + x_0\big(a_{n-1} + b_n x_0\big)\ldots\Big)\Big) \\
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\Big(a_2 + \cdots + x_0\big(b_{n-1}\big)\ldots\Big)\Big) \\
p(x_0) &= a_0 + x_0(b_1) \\
p(x_0) &= b_0
\end{aligned}
$$

Hornerovo schéma – ruční výpočet

Vypočtěte hodnotu polynomu $p(x) = 2x^3 - 6x^2 + 2x - 1$ v bodě $x_0 = 3$.

x_0	x^3	x^2	x^1	x^0
3	2	-6	2	-1
		6	0	6
	2	0	2	5

Běžný výpočet

$$\begin{aligned} p(3) &= 2 \times 3^3 - 6 \times 3^2 + 2 \times 3 - 1 \\ &= 2 \times 27 - 6 \times 9 + 2 \times 3 - 1 \\ &= 54 - 54 + 6 - 1 = 5 \end{aligned}$$

Hornerovo schéma

ALGORITHM $Horner(P[0..n], x)$

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n-1$ **downto** 0 **do**
    $p \leftarrow x * p + P[i]$
return p

Hornerovo schéma – časová složitost algoritmu

Je zřejmé, že počet násobení $M(n)$ a počet sčítání $A(n)$ je roven

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$$

Výpočet hrubou silou

Jen pro výpočet $a_n x^n$ je zapotřebí:

- $n - 1$ násobení pro výpočet mocniny
- 1 násobení pro vynásobení a_n.

Za shodný počet násobení zvládne Hornerův algoritmus vypočítat i zbývajících $n - 1$ členů polynomu!!!

Zdroje pro samostatné studium

- Kniha [2], kapitola 6.5, strany 234 – 239
- Kniha [3], kapitola 30.1, strany 879 – 880

Strategie řešení transformuj a vyřeš

Redukce problému

Redukce problému

Smyslem redukce je převedení řešeného problému na problém jiný, který umíme vyřešit.

Postup redukce:

1. **Problém 1** – to, co chceme řešit
2. Redukce **Problému 1** na **Problém 2**
3. **Problém 2** – řešitelný algoritmem A
4. Provedení algoritmu A
5. **Řešení Problému 2**

Nejmenší společný násobek

Nejmenší společný násobek $lcm(m, n)$ dvou přirozených čísel m a n definujeme jako nejmenší přirozené číslo, které je dělitelné m a n zároveň.

Řešení pomocí prvočíselného rozkladu:

$$\begin{aligned} 24 &= 2^3 \cdot 3^1 \\ 60 &= 2^2 \cdot 3^1 \cdot 5^1 \\ \operatorname{lcm}(24, 60) &= 2^3 \cdot 3^1 \cdot 5^1 = 120 \end{aligned}$$

Řešení pomocí největšího společného dělitele:

Lze dokázat, že

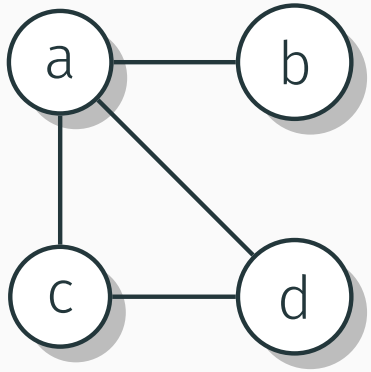
$$\operatorname{lcm}(m, n) = \frac{mn}{\gcd(m, n)}$$

$\gcd(m, n)$ lze vypočítat, efektivním, Euklidovým algoritmem.

Počet sledů v grafu

Zadání: Vypočítat počet sledů mezi dvojicemi vrcholů v daném grafu G.

Řešení: Lze dokázat, že počet různých sledů délky k mezi vrcholy i a j je roven prvku a_{ij} matice $\mathbf{A}^k$, kde A je matice sousednosti grafu G.



$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad \mathbf{A}^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \end{array}$$

Z a do a vedou tři sledy délky 2: $a - b - a$, $a - c - a$, $a - d - a$

Z a do c vede jeden sled délky 2: $a - d - c$

Redukce optimalizačních problémů

Maximalizační problém – nalezení maxima funkce $f(x)$

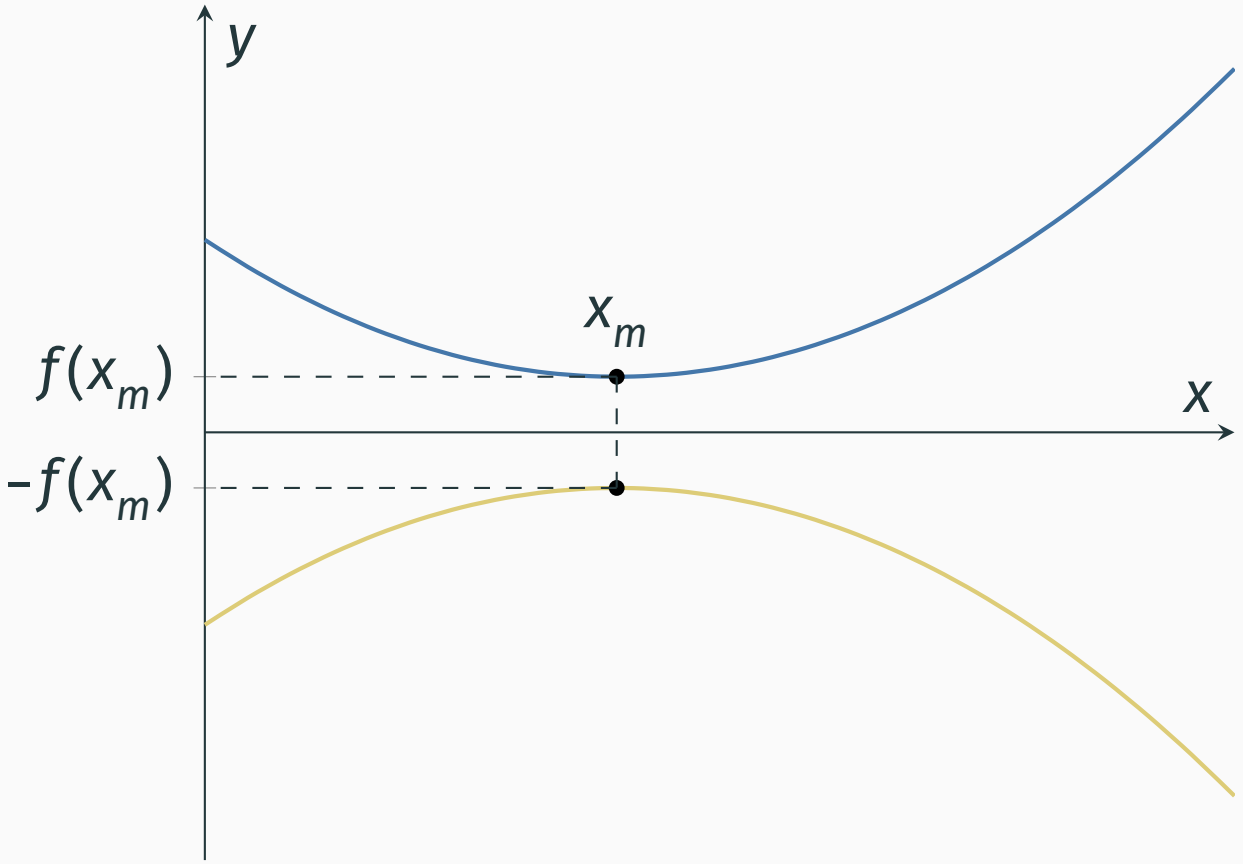
Minimalizační problém – nalezení minima funkce $f(x)$

Jak řešit situaci?

- Máme minimalizovat funkci $f(x)$, ale
- k dispozici máme pouze maximalizační algoritmus.

Lze využít maximalizační algoritmus pro minimalizační problém? Případně naopak?

Redukce optimalizačních problémů



$$\min f(x) = -\max\left[-f(x)\right]$$
$$\max f(x) = -\min\left[-f(x)\right]$$

Koza, vlk a zelí

- Na břehu řeky je převozník, koza, vlk a zelí.
- Převozník má převézt kozu, vlka a zelí na druhý břeh pomocí loďky.
- Na loďku se, mimo převozníka, vejde nejvýše jedna z převážených entit.
- Na stejném břehu se nesmí bez převozníkova dozoru ocitnout dvojice (koza, zelí) a (vlk, koza).
- Úkolem je sestavit plán převozu nebo dokázat, že řešení neexistuje.

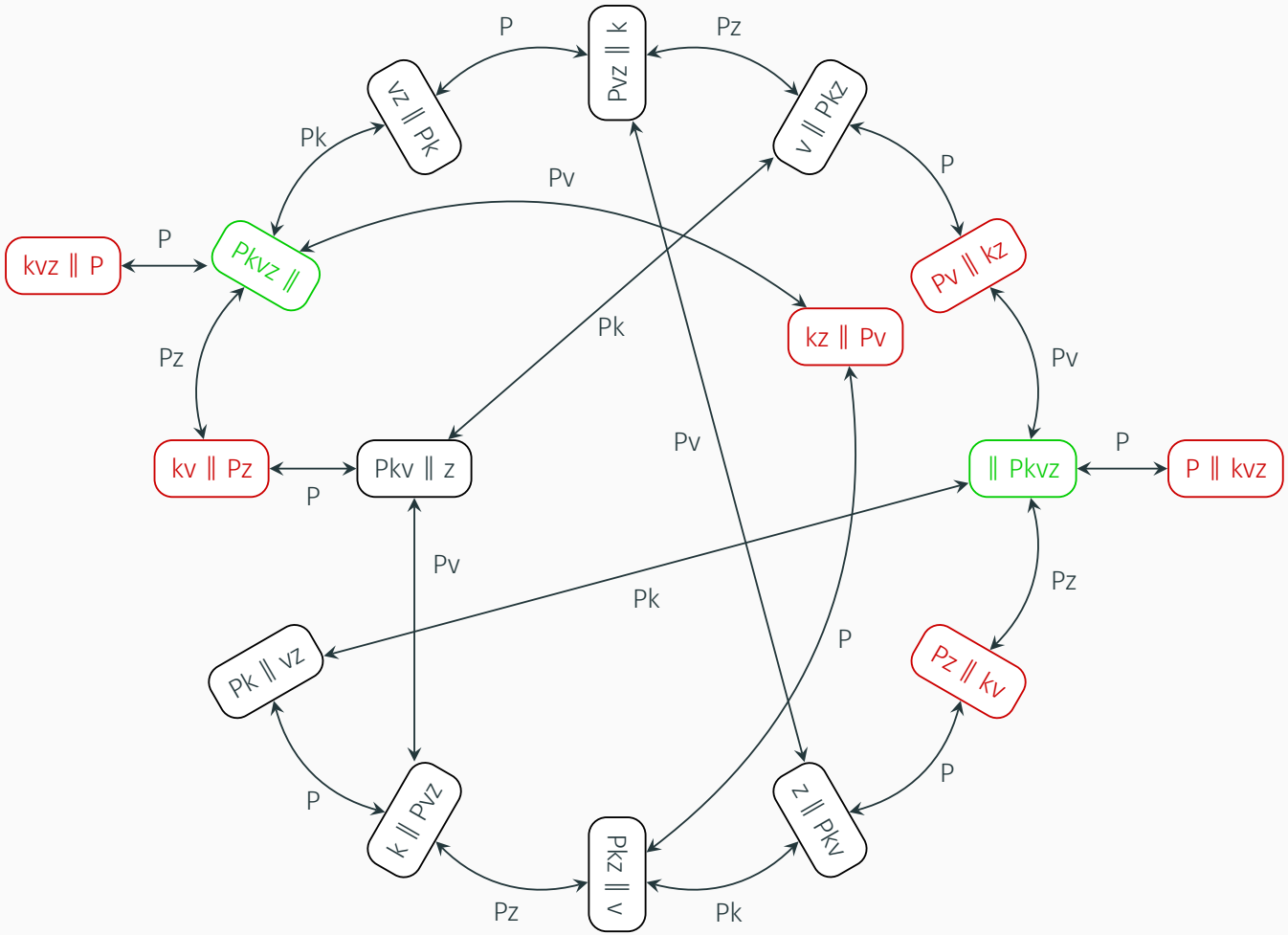
Nejstarší písemná podoba úlohy pochází z 9. století…

Koza, vlk a zelí – stavový prostor

Stav – reprezentuje obsazení obou břehů oddělených řekou, např. Pkv||z

Přechod mezi stavy – cesta z jednoho břehu řeky na druhý, s případným převozem

Koza, vlk a zelí – graf stavového prostoru



Řešení problému – nalezení orientované cesty z počátečního stavu do koncového stavu průchodem do šířky.

Zdroje pro samostatné studium

- Kniha [2], kapitola 6.6, strany 240 – 248

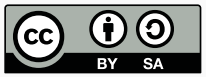
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Záměna paměťové a časové složitosti

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Záměna paměťové a časové složitosti

B-stromy

B-stromy – motivace

- Zpracování velkého množství strukturovaných záznamů (lze je identifikovat jednoznačným klíčem), které přesahuje dostupnou operační paměť.
- Data musí být uložena ve vnější paměti, tzv. „na disku“.
- Disk nabízí jen sekvenční soubor.
- Hledáme datovou strukturu, která umožní v takovém souboru efektivně vyhledávat, vkládat a mazat záznamy.
- Odpovědí je záměna paměťové složitosti za časovou, jinak řečeno zvýšíme paměťovou složitost (obětujeme extra paměť navíc), abychom snížili časovou složitost operací.

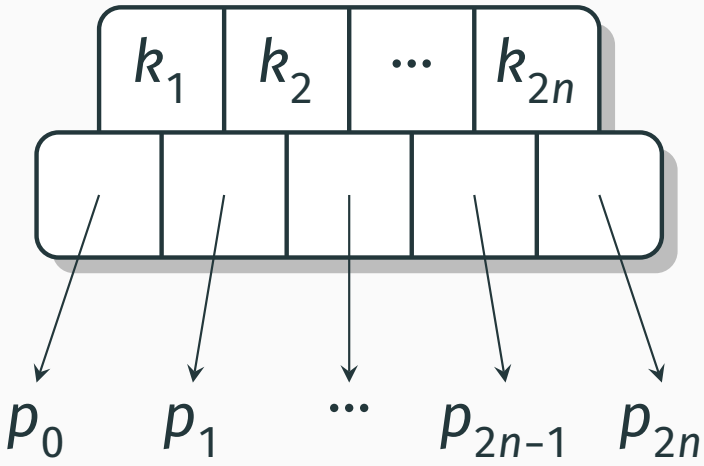
B-stromy

B-strom řádu n je $(2n + 1)$-ární strom, který splňuje následující kritéria:

1. Každá stránka obsahuje nejvýše $2n$ klíčů.
2. Každá stránka, s výjimkou kořenové obsahuje alespoň n klíčů.
3. Každá stránka je buď listovou stránkou, tj. nemá žádné potomky, nebo má $m + 1$ potomků, kde m je aktuální počet klíčů ve stránce.
4. Všechny listové stránky jsou na stejné úrovni. Jinak řečeno strom je dokonale vyvážený.

Publikováno Rudolfem Bayerem v roce 1972 [8].

B-stromy – schéma stránky

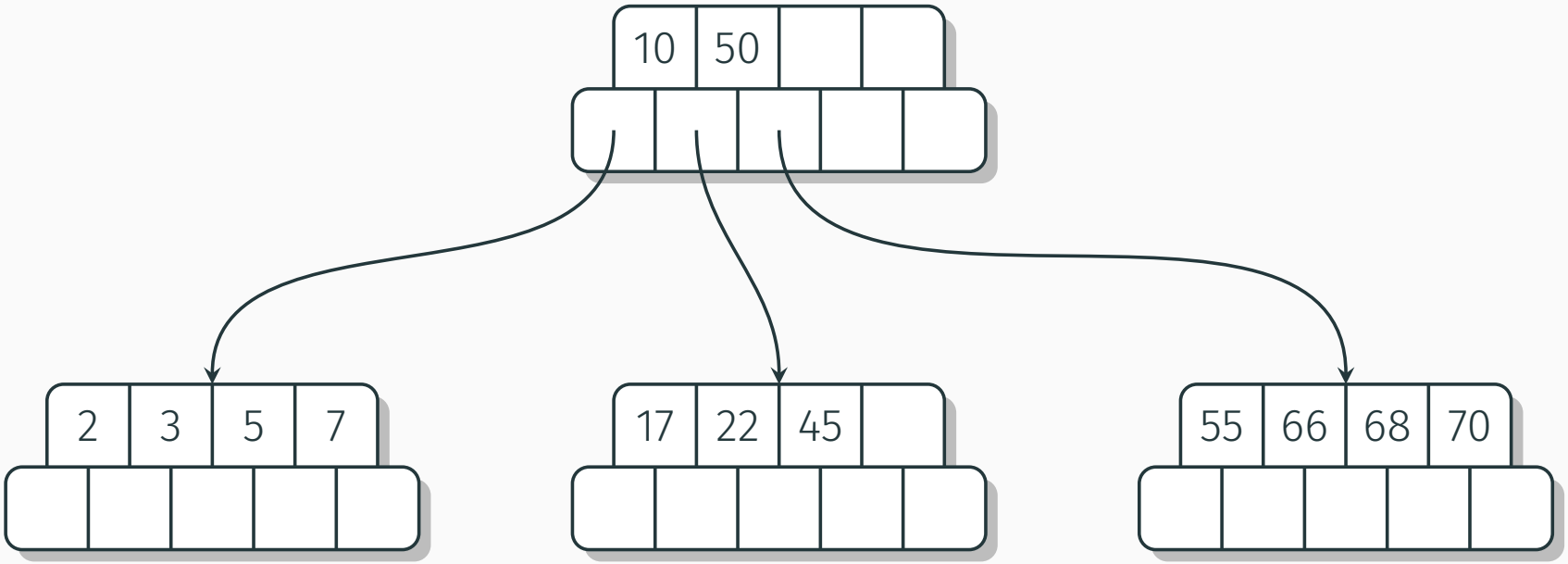


- Uzly v B-stromu se tradičně nazývají **stránky** (pages).
- Počet klíčů ve stránce kolísá od n do $2n$, výjimku tvoří kořen stromu.
- Klíče ve stránce jsou setříděné, tedy $k_1 \leq k_2 \leq \cdots \leq k_{2n}$.
- Pro klíče v podstromech na které odkazují ukazatelé $p_0, \ldots, p_{2n}$ platí

$$K_{p_0} \leq k_1 \leq K_{p_1} \leq k_1 \leq \cdots K_{p_{2n-1}} \leq k_{2n} \leq K_{p_{2n}},$$

kde K_{p_i} je množina všech klíčů v podstromu s kořenem ve stránce na kterou ukazuje p_i.

B-stromy – ukázka



B-stromy – poznámky

- Z definice je zřejmé, že B-strom není vždy zcela zaplněn. Faktor zaplnění kolísá od 50 % do 100 %.
- Volné místo ve stromu umožňuje snadné vkládání dalších klíčů.
- Díky stromové struktuře lze operace vyhledávání, vkládání a mazání klíčů v B-stromu provádět s logaritmickou časovou složitostí.
- Algoritmy pro B-stromy jsou zobecněním algoritmů pro binární vyhledávací stromy.

B-stromy – alternativní definice, varianty

- Výše uvedená definice připouští jen B-stromy s max. kapacitou $2n$ klíčů, tj. sudým číslem.
- Max. kapacita může být ale libovolná, i lichá.
- Některé definice označují pomocí čísla n max. kapacitu.
- Počet klíčů ve stránce tak kolísá od $\left\lceil \frac{n}{2} \right\rceil$ do n.
- Cílem naší definice je snadná pochopitelnost principů fungování B-stromu a jednoduchý zápis.
- Někdy se lze setkat i s definicí, kde číslo n označuje max. počet potomků nikoliv počet klíčů ve stránce.

B-stromy – varianty

B^+-strom

- všechny klíče jsou uloženy pouze v listech
- listy jsou navzájem propojeny odkazy – rychlejší práce se souvislými úseky klíčů, „najdi všechny klíče mezi 100 a 200“
- v knize od Levitina [2] je jako B-strom popsána právě tato varianta.

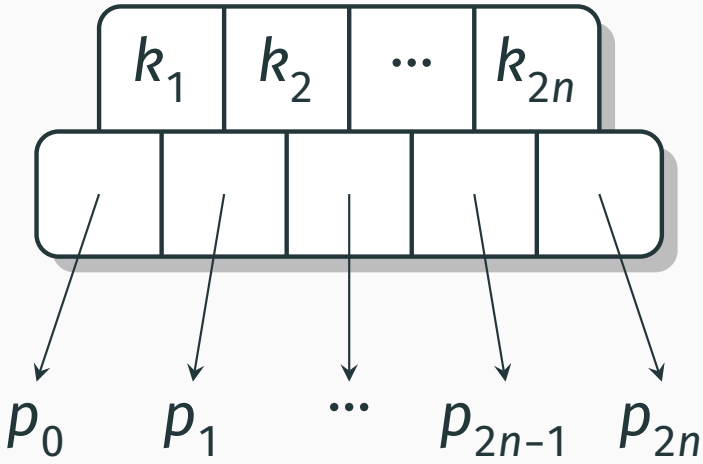
B^*-strom

- stránka musí být zaplněna minimálně ze dvou třetin,
- při vkládání klíče do zaplněné stránky jsou klíče nejprve přesouvány mezi sourozenci,
- dochází k menšímu počtu štěpení stránek.

B-stromy – vyhledávání klíče x

1. Na počátku algoritmu označíme kořenovou stránku za aktuální stránku P.
2. Pokud stránka P neexistuje, vyhledávání končí neúspěchem.
3. Jinak předpokládejme, že ve stránce P je m klíčů $k_1, \ldots, k_m$ a odpovídající ukazatelé na potomky $p_0, \ldots p_m$. Potom:
   - 3.1 Pokud $x = k_i$, pro některé $1 \leq i \leq m$, pak vyhledání končí úspěchem.
   - 3.2 Jestliže $x < k_1$, potom $P = p_0$ a zpět k bodu 2.
   - 3.3 Jestliže $x > k_m$, potom $P = p_m$ a zpět k bodu 2.
   - 3.4 Jinak nalezneme takové i, $1 \leq i \leq m$, pro které platí, že $k_i < x < k_{i+1}$. Potom $P = p_i$ a a zpět k bodu 2.

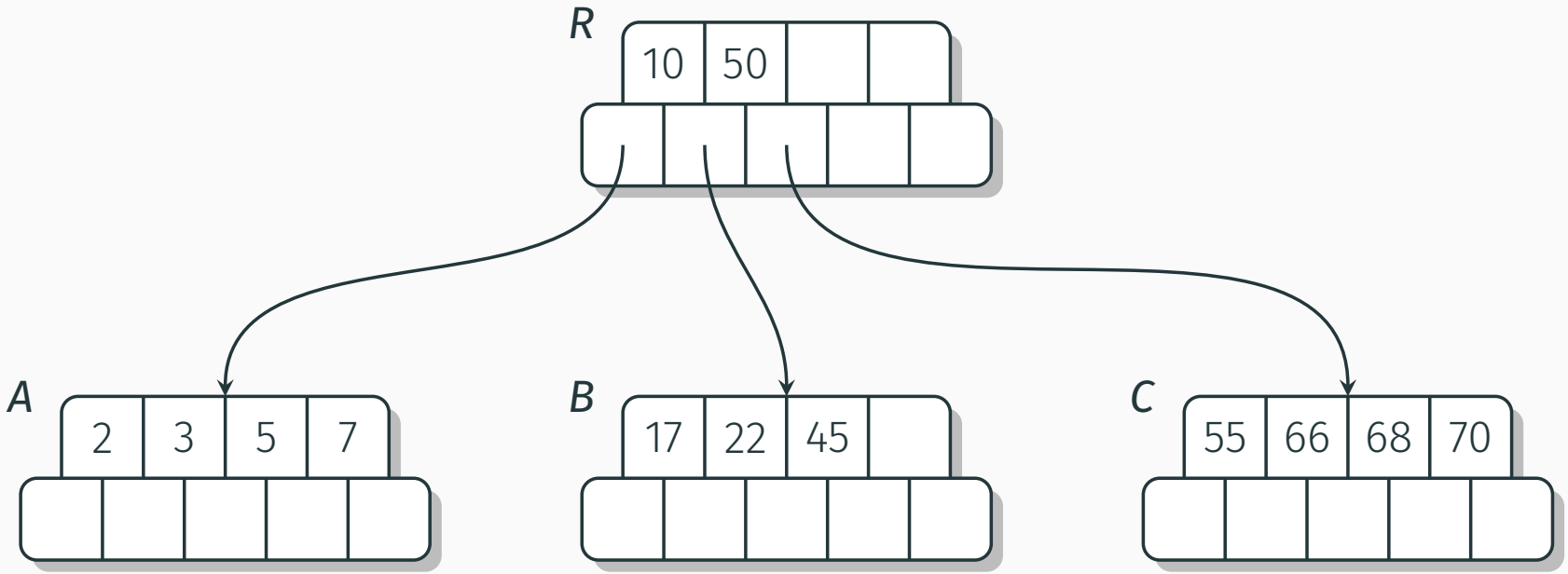
Příklady operací s B-stromem



V ukázkách budeme používat B-strom pro $n = 2$, tzn. každá stránka obsahuje minimálně 2 a maximálně 4 klíče.

A dále každá stránka odkazuje na minimálně 3 a maximálně 5 potomků. Výjimku tvoří kořen stromu.

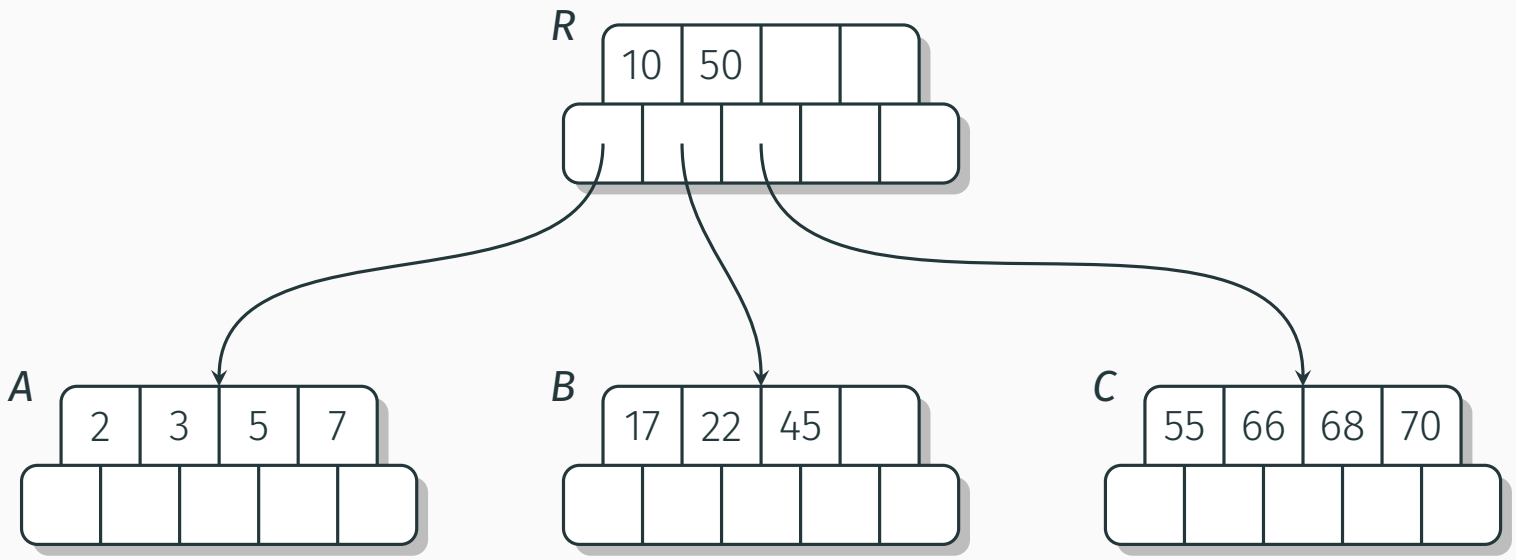
Příklad vyhledávání v B-stromu – vyhledání klíče 50



Postup

1. Vyhledávání zahájíme v kořeni R, tedy $P = R$.
2. Stránka P existuje, pokračujeme dalším bodem.
3. Protože $x = k_2$ hledání končí úspěchem.

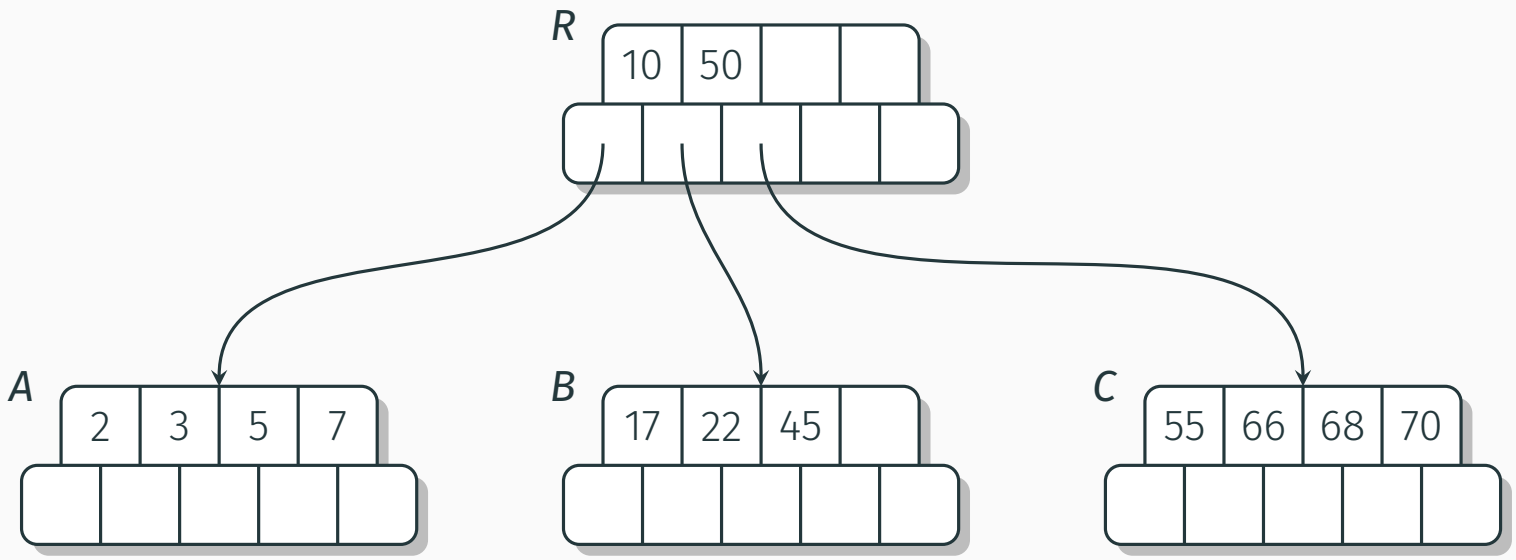
Příklad vyhledávání v B-stromu – vyhledání 3



Postup

1. Vyhledávání zahájíme v kořeni R, tedy $P = R$.
2. Stránka P existuje, pokračujeme dalším bodem.
3. Protože $x < k_1$, pak $P = p_0 = A$.
4. Stránka P existuje, pokračujeme dalším bodem.
5. Protože $x = k_2$ hledání končí úspěchem.

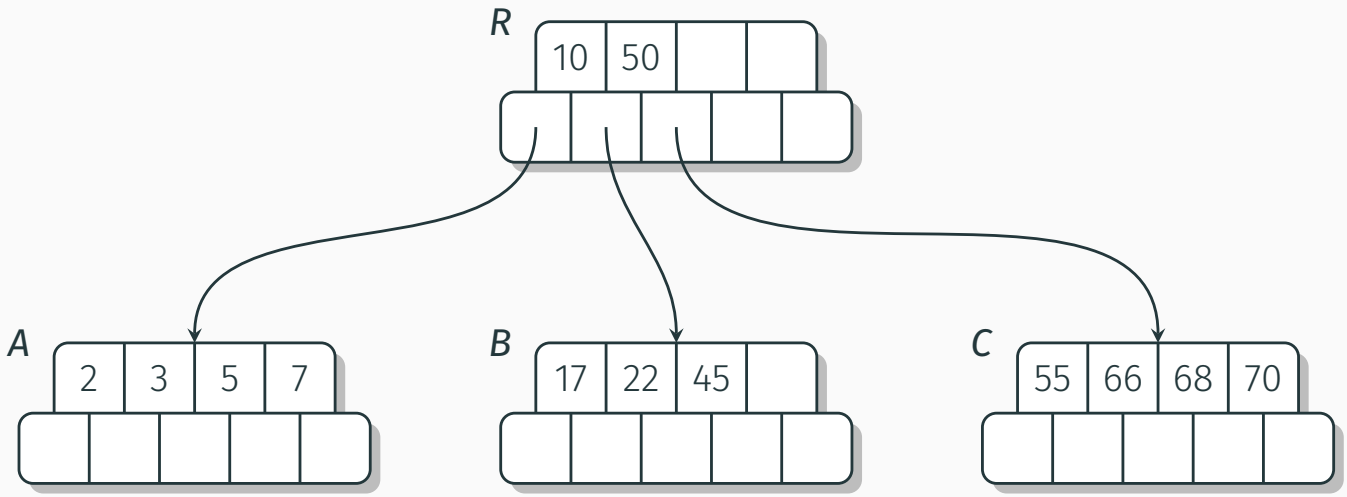
Příklad vyhledávání v B-stromu – vyhledání 45



Postup

1. Vyhledávání zahájíme v kořeni R, tedy $P = R$.
2. Stránka P existuje, pokračujeme dalším bodem.
3. Protože $k_1 < x < k_2$, pak $P = p_1 = B$.
4. Stránka P existuje, pokračujeme dalším bodem.
5. Protože $x = k_3$ hledání končí úspěchem.

Příklad vyhledávání v B-stromu – vyhledání 57



Postup

1. Vyhledávání zahájíme v kořeni R, tedy $P = R$.
2. Stránka P existuje, pokračujeme dalším bodem.
3. Protože $x > k_2$, pak $P = p_2 = C$.
4. Stránka P existuje, pokračujeme dalším bodem.
5. Protože $k_1 < x < k_2$, pak $P = p_1 = null$.
6. Protože P neexistuje, hledání končí **neúspěchem**.

B-stromy – vložení klíče x

1. Nejprve je nutné určit, pomocí algoritmu vyhledávání, listovou stránku L kam bude klíč x vložen.
2. Mohou nastat dva případy:
   - Stránka L není zcela zaplněna – klíč x je vložen do stránky tak, aby zůstalo zachováno uspořádání klíčů.
   - Stránka L je zcela zaplněna, potom
     2.1 klíč x zatřídíme (např. v pomocném poli) mezi klíče ze stránky L tak, aby zůstalo zachováno uspořádání klíčů. Dostaneme posloupnost $2n + 1$ klíčů $k'_1 < k'_2 < \cdots k'_{2n+1}$
     2.2 vytvoříme novou stránku P, se stejným rodičem R jako L
     2.3 distribuce klíčů do stránek

Klíče	Akce
$k'_1, \ldots, k'_n$	ponechat ve stránce L
k'_n	vložit do rodičovské stránky R
$k'_{n+2}, \ldots k'_{2n+1}$	vložit do nové stránky P

B-stromy – algoritmus vkládání, poznámky

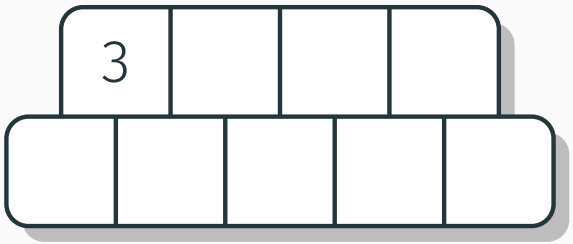
- Proces vytvoření nové stránky a přerozdělení klíčů nazýváme **štěpení stránky**.
- Vložením klíče k'_n do rodičovské stránky R dojde ke zvýšení počtu klíčů v této stránce a tím i k nárůstu počtu odkazů na potomky této stránky. Bez přesunu k'_n by ve stránce R chyběl volný odkaz pro připojení stránky P.
- Vložení klíče k'_n do stránky R se realizuje stejným algoritmem jako vložení klíče x do L. Vložení k'_n může způsobit rozštěpení stránky R.
- Štěpení stránek může vést až k vytvoření nového kořene celého stromu, což je jediný způsob jak B-strom může zvýšit svoji výšku.

Příklad vkládání do B-stromu

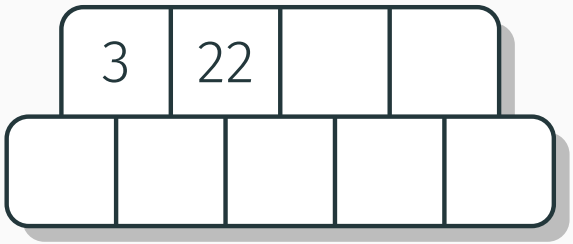
- V tomto rozsáhlejším ukázkovém příkladu budeme postupně budovat B-strom se stejnými parametry jako u příkladu vyhledávání.
- Do stromu postupně vložíme klíče 3, 22, 10, 2, 17, 5, 66, 68, 50, 7, 55, 45, 70, 44, 6, 21, 67, 1, 4, 8, 9, 12 a 15.

Příklad vkládání do B-stromu – vložení klíčů 3, 22 a 10

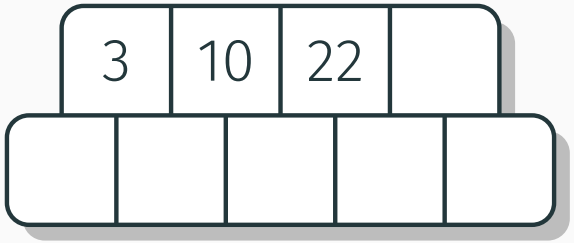
Vložení klíče 3



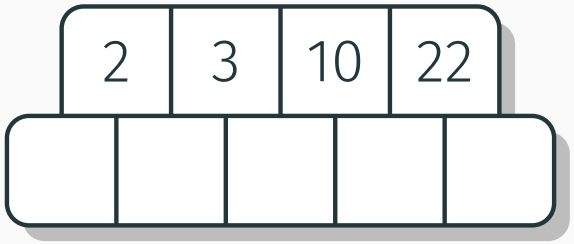
Vložení klíče 22



Vložení klíče 10

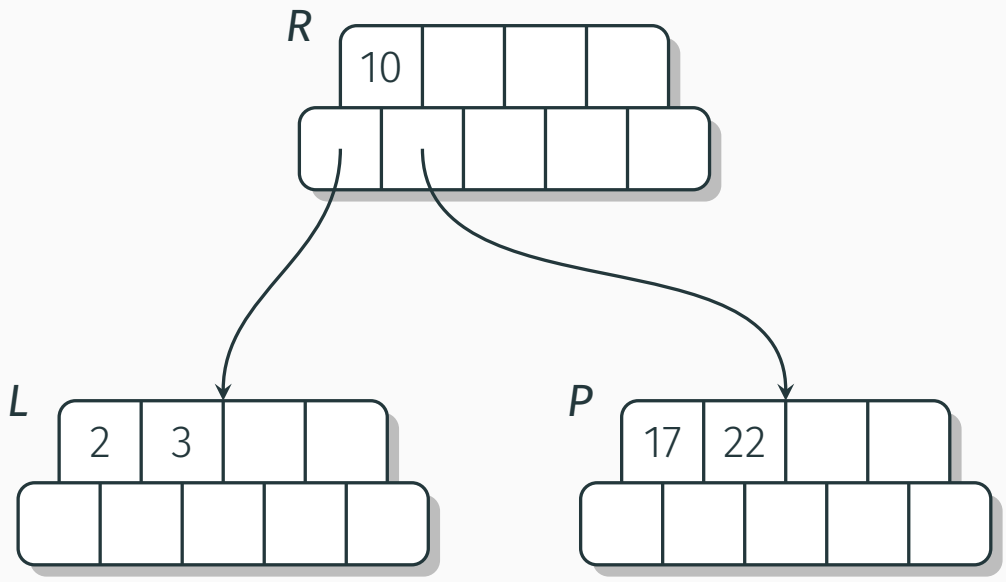


Příklad vkládání do B-stromu – vložení klíče 2



Stránka je zcela zaplněna, vložení libovolného dalšího klíče způsobí změnu struktury B-stromu.

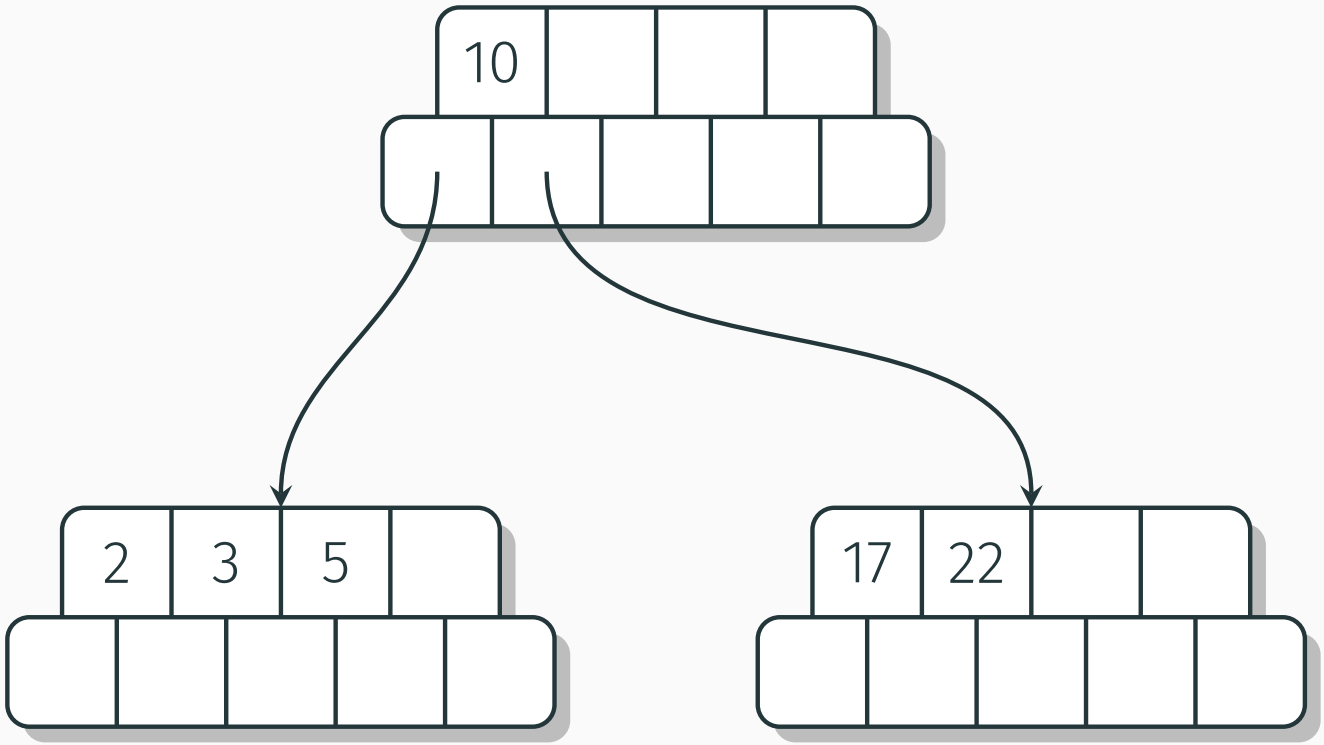
Příklad vkládání do B-stromu – vložení klíče 17



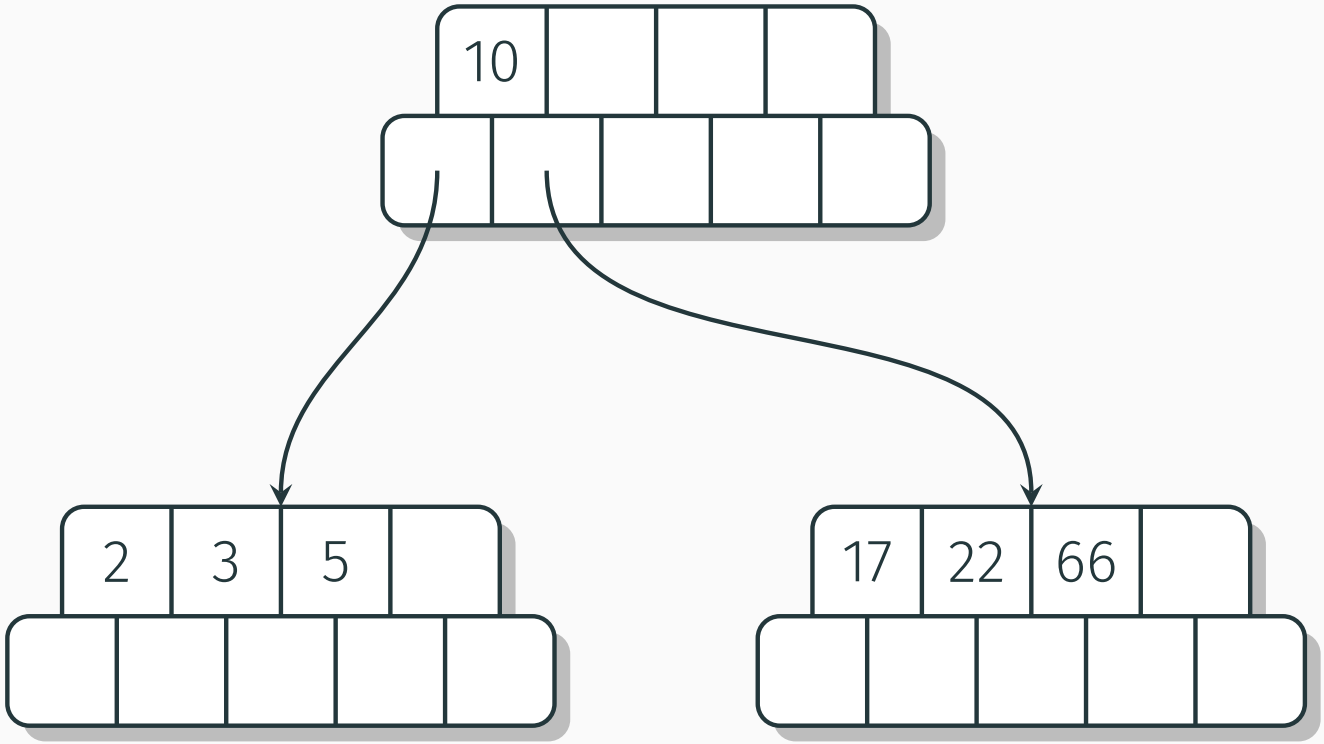
Vložením klíče 17 došlo:

1. k rozštěpení stránky *L* a k přesunu poloviny klíčů do nové stránky *P*,
2. ke vzniku nové kořenové stránky *R* a k přesunu klíče 10 do nového kořene.

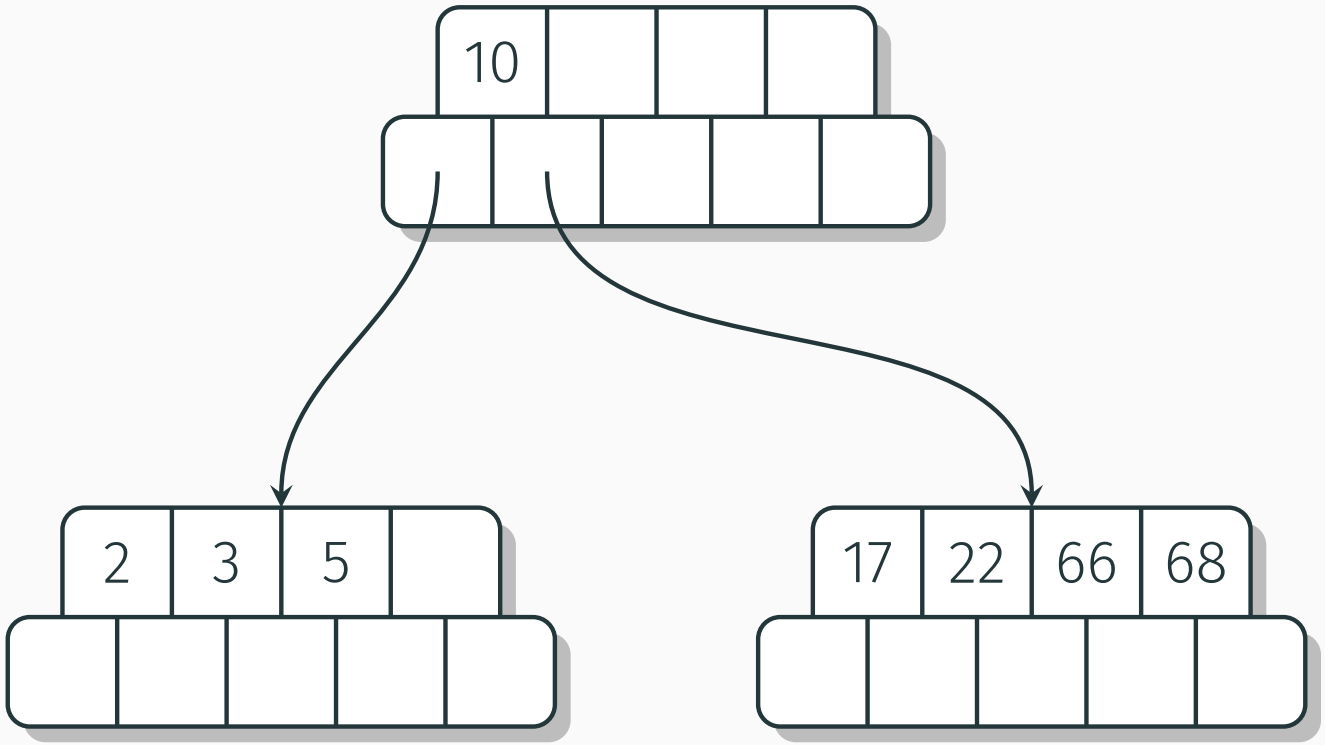
Příklad vkládání do B-stromu – vložení klíče 5



Příklad vkládání do B-stromu – vložení klíče 66

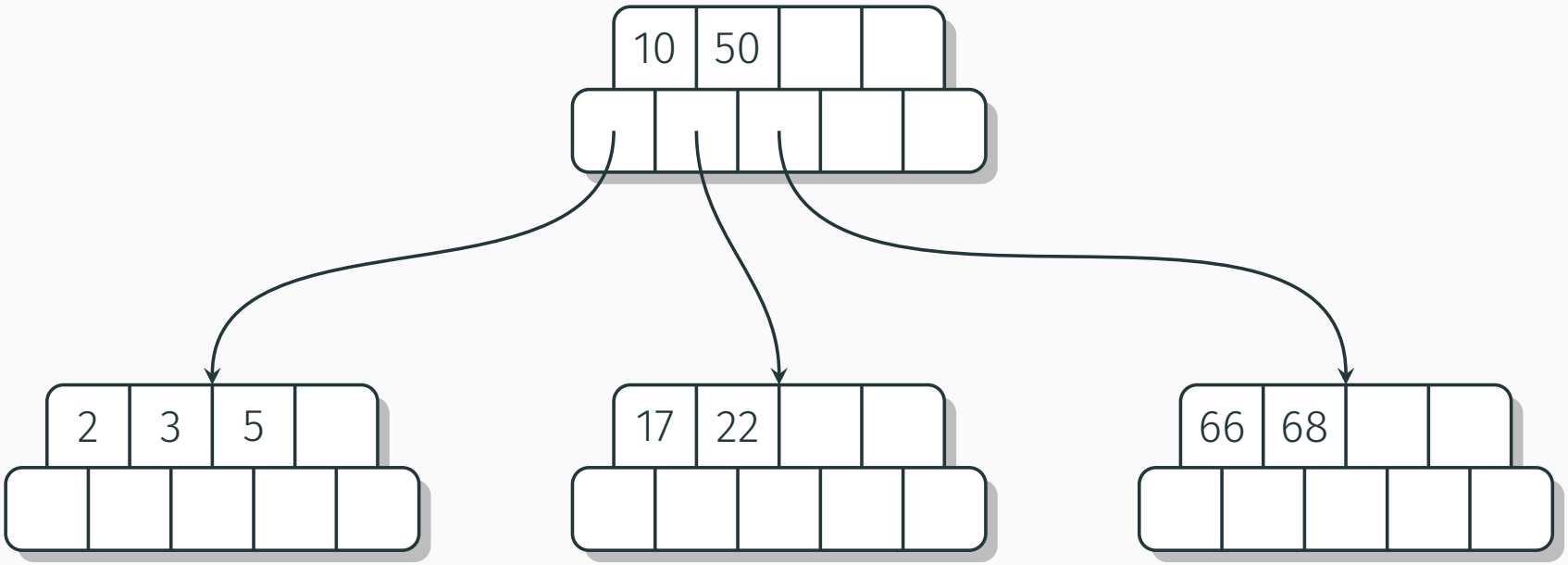


Příklad vkládání do B-stromu – vložení klíče 68



Stránka s klíči 17 až 68 je zcela zaplněna, vložení dalšího klíče do této stránky způsobí změnu struktury B-stromu.

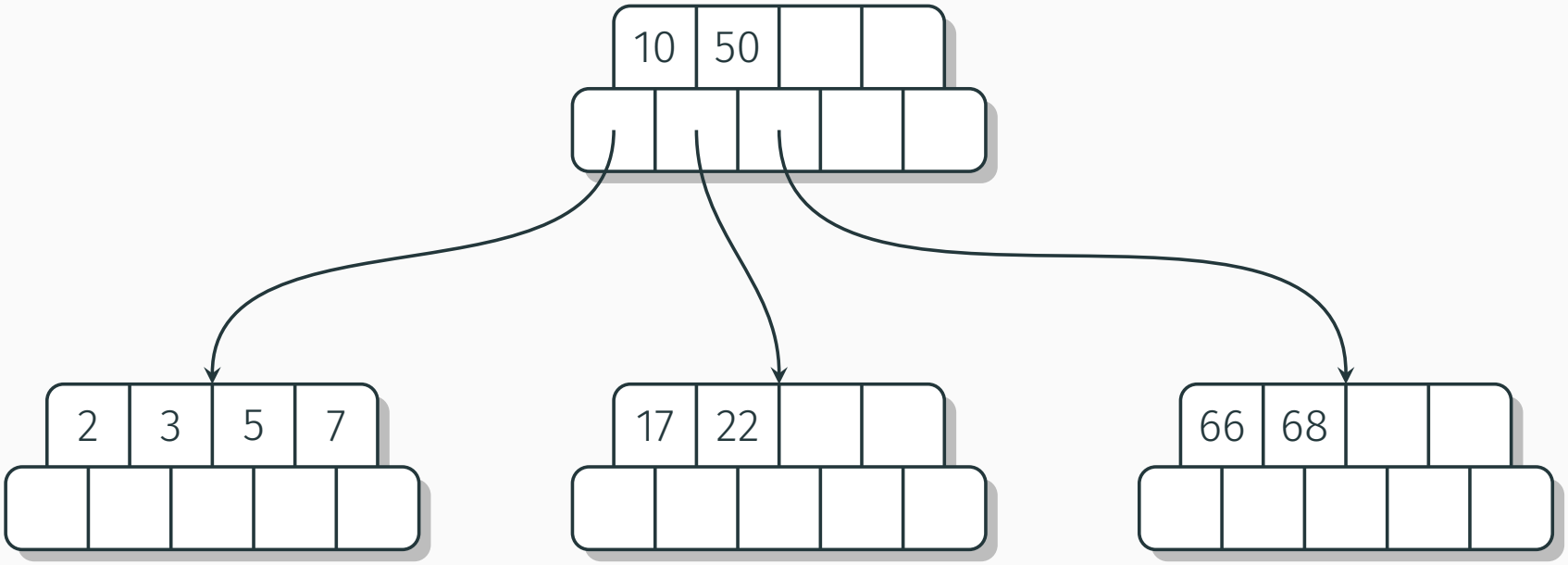
Příklad vkládání do B-stromu – vložení klíče 50



Vložením klíče 50 došlo:

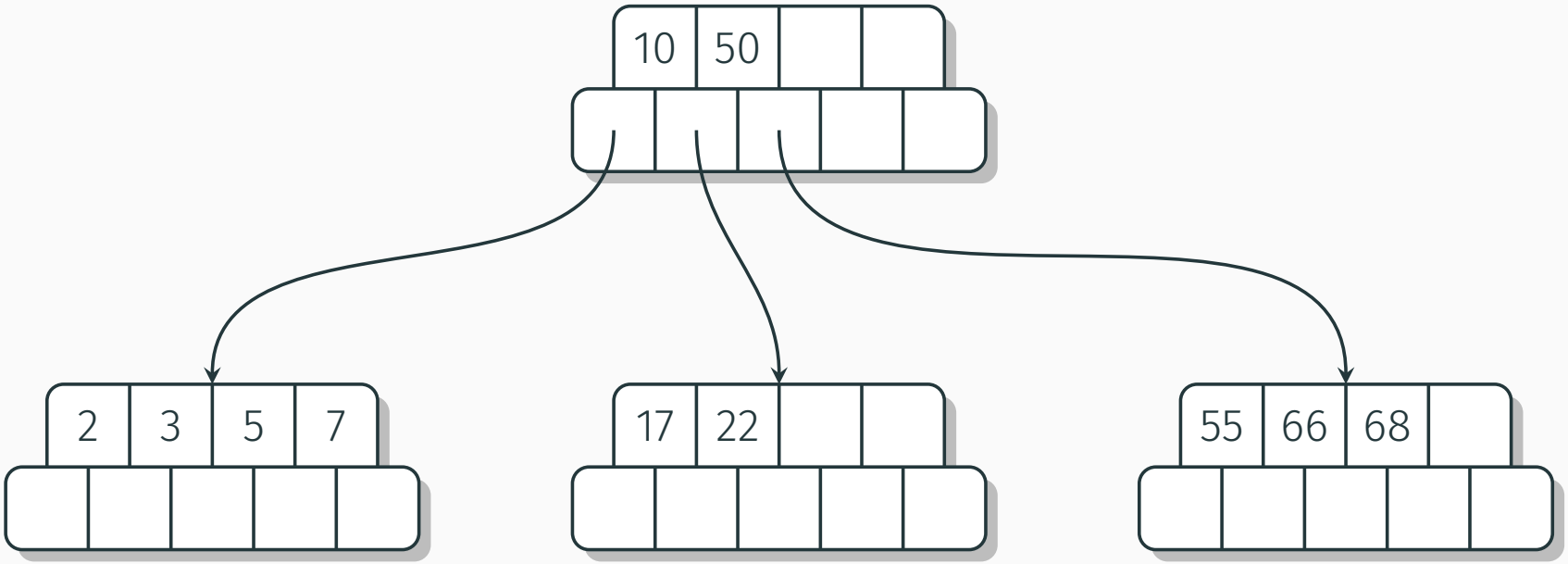
1. k rozštěpení stránky a k přesunu poloviny klíčů do nové stránky a
2. současně byl nově vložený klíč 50, jakožto medián hodnot v původní stránce, přesunut do kořenové stránky.

Příklad vkládání do B-stromu – vložení klíče 7

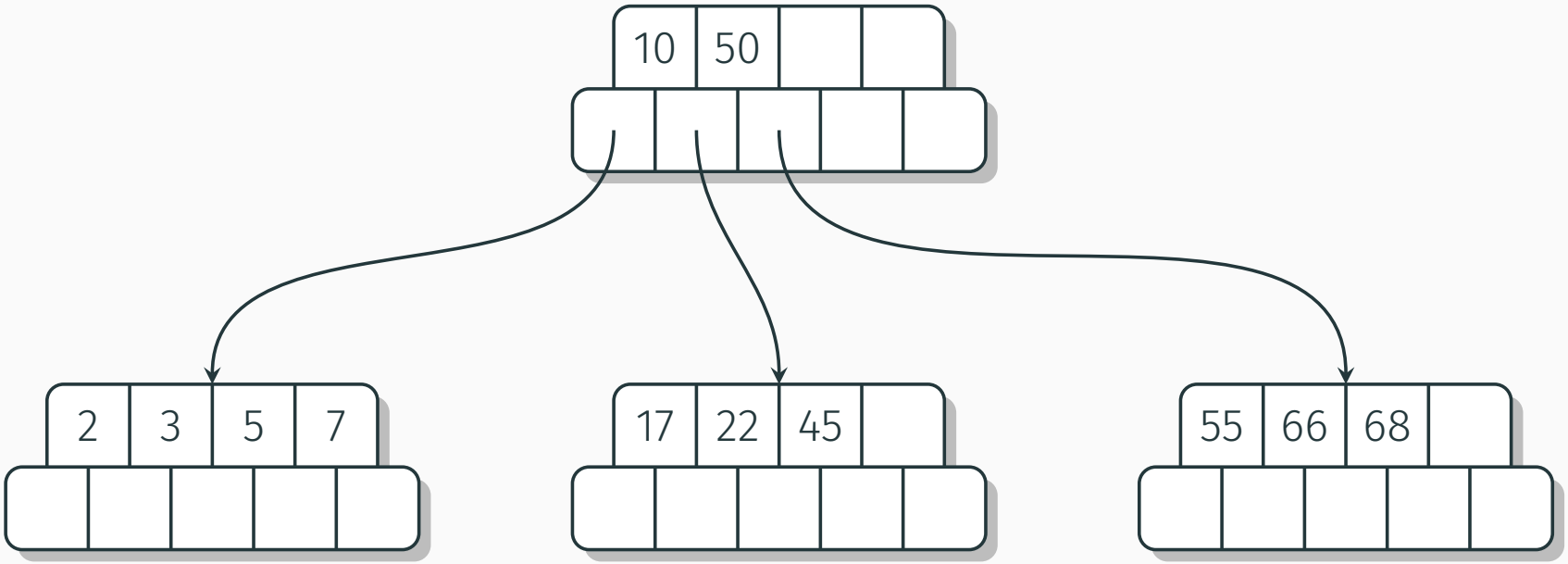


Stránka s klíči 2 až 7 je zcela zaplněna, vložení dalšího klíče do této stránky způsobí změnu struktury B-stromu.

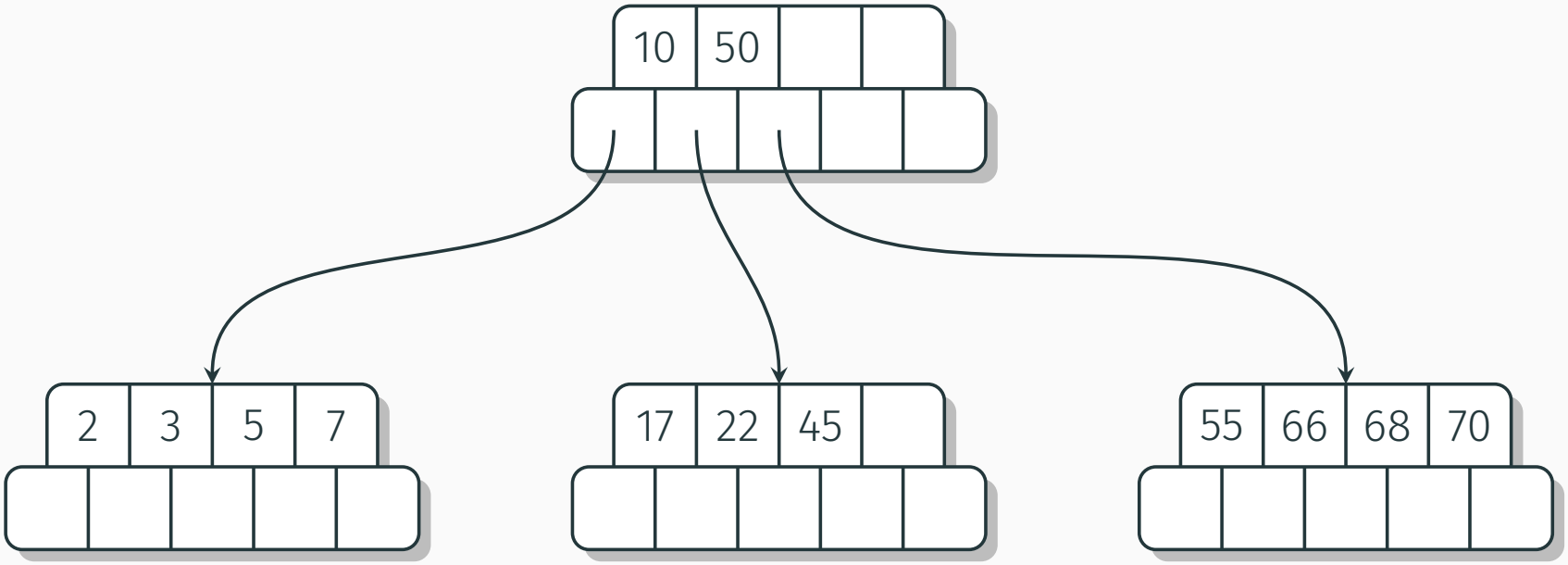
Příklad vkládání do B-stromu – vložení klíče 55



Příklad vkládání do B-stromu – vložení klíče 45

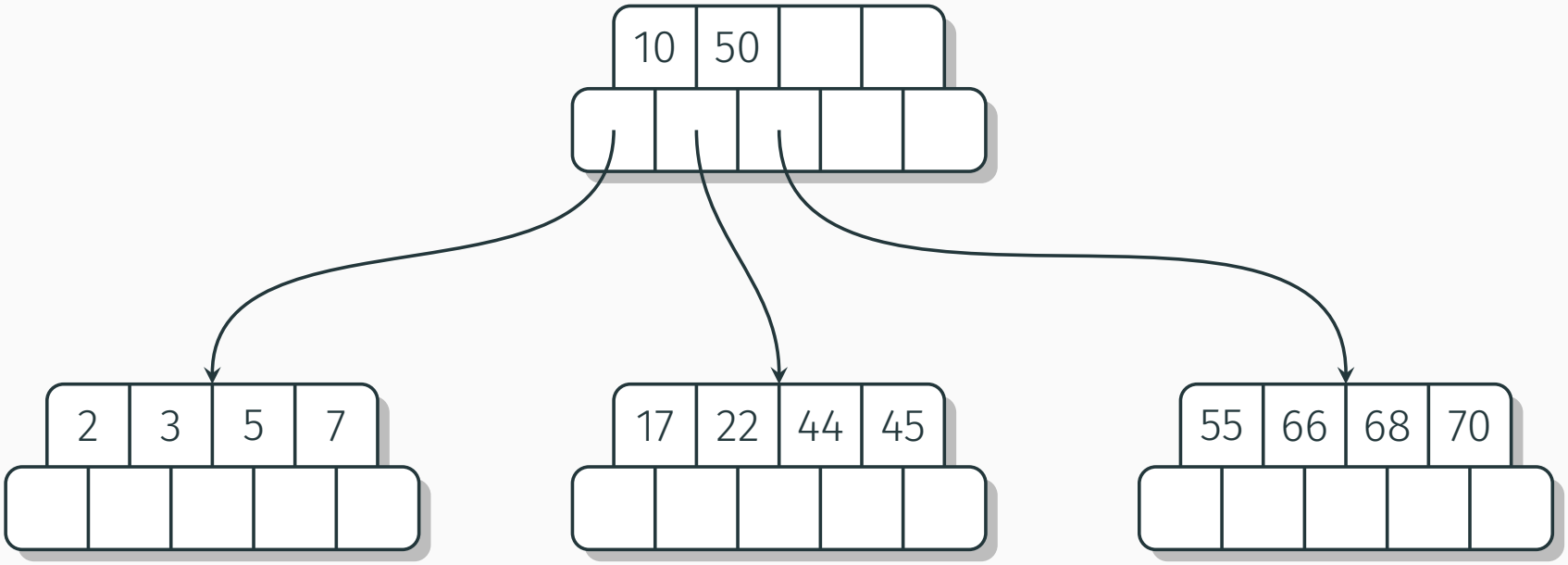


Příklad vkládání do B-stromu – vložení klíče 70



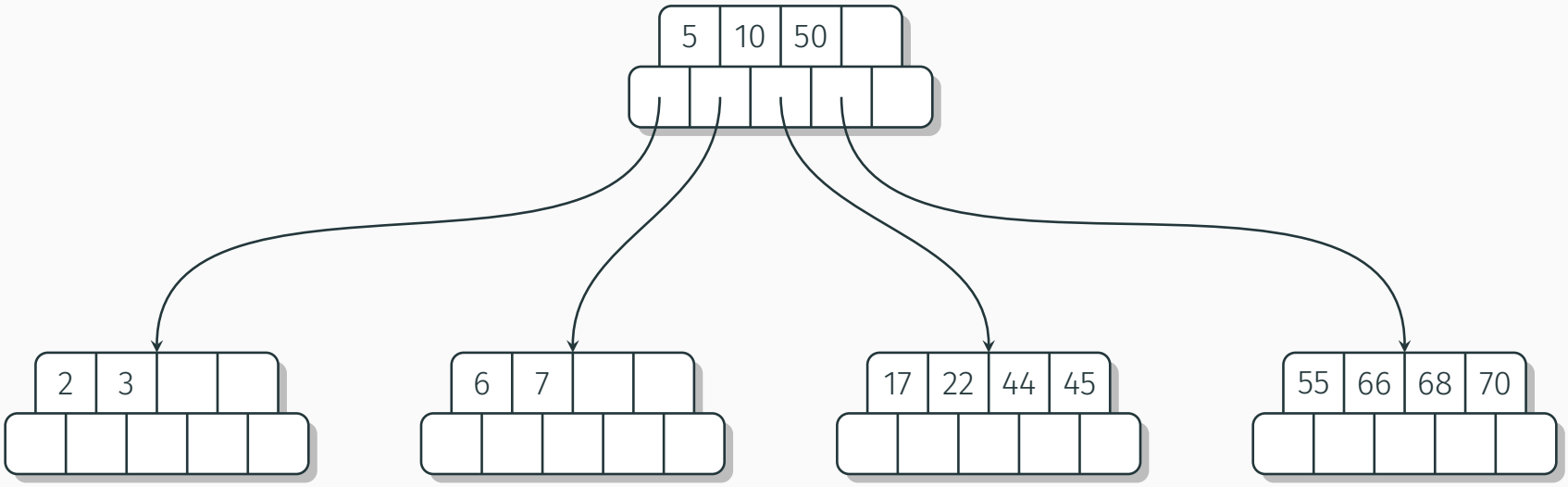
Stránka s klíči 55 až 70 je zcela zaplněna, vložení dalšího klíče do této stránky způsobí změnu struktury B-stromu.

Příklad vkládání do B-stromu – vložení klíče 44



Všechny listové stránky jsou zcela zaplněny, vložení jakéhokoliv dalšího klíče způsobí změnu struktury B-stromu.

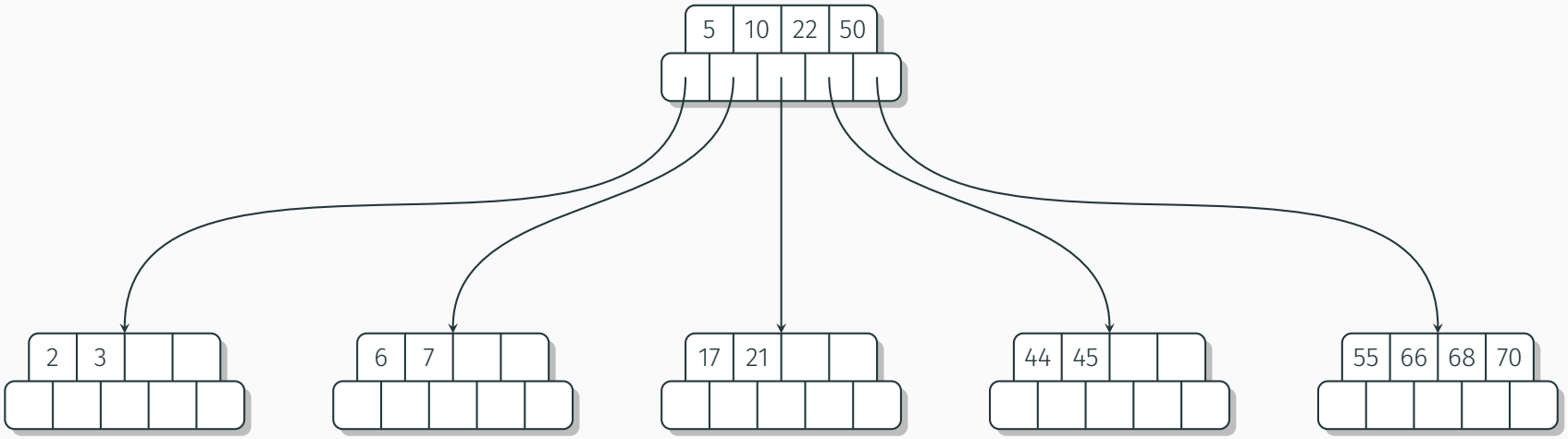
Příklad vkládání do B-stromu – vložení klíče 6



Vložením klíče 6 došlo:

1. k rozštěpení stránky a k přesunu poloviny klíčů do nové stránky a
2. současně k přesunu klíče 5 do kořenové stránky.

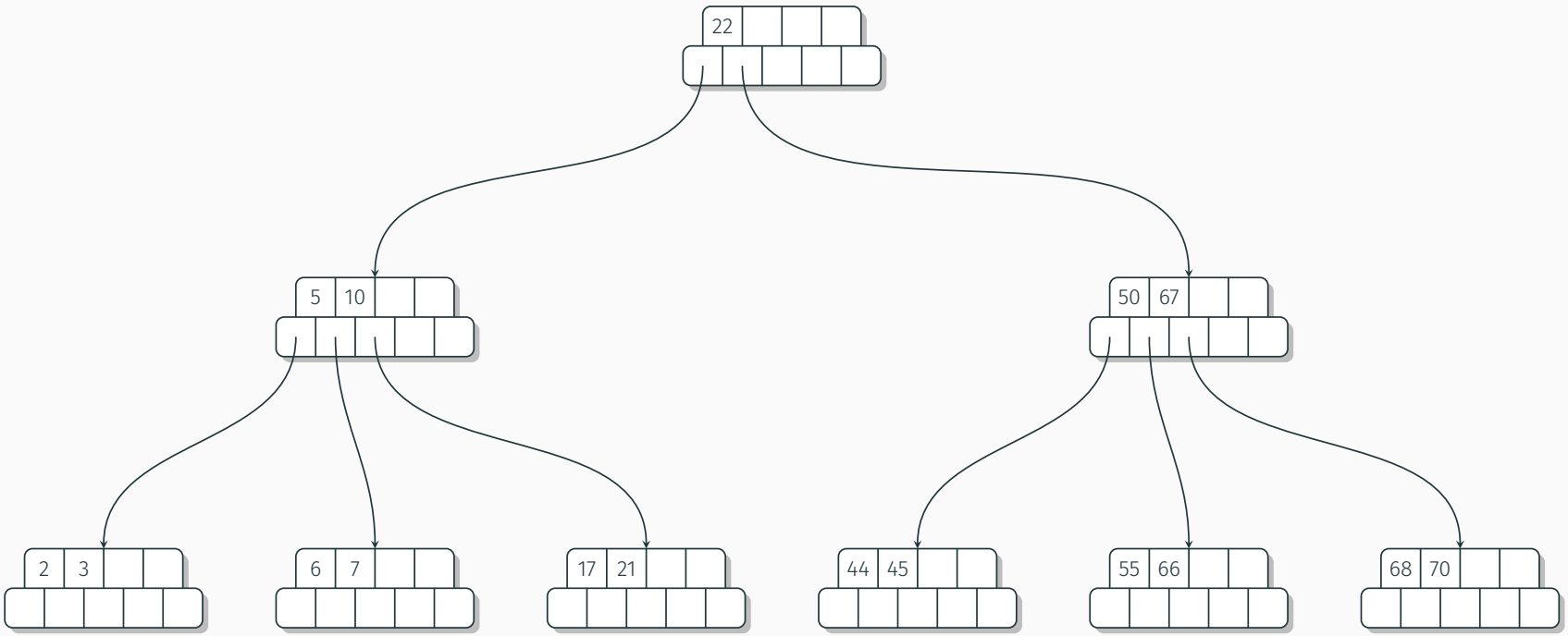
Příklad vkládání do B-stromu – vložení klíče 21



Vložením klíče 21 došlo:

1. k rozštěpení stránky a k přesunu poloviny klíčů do nové stránky a
2. k přesunu klíče 22 do rodičovské stránky.
3. Zároveň došlo k zaplnění kořenové stránky stromu.

Příklad vkládání do B-stromu – vložení klíče 67



Příklad vkládání do B-stromu – vložení klíče 67 (pokrač.)

Vložením klíče 67 došlo:

1. k rozštěpení stránky a k přesunu poloviny klíčů do nové stránky a
2. současně byl nově vložený klíč 67, jakožto medián hodnot v původní stránce, přesunut do rodičovské stránky.
3. Protože i tato stránka byla již zcela zaplněna, došlo k jejímu rozštěpení a tím pádem k vytvoření nové kořenové stránky stromu s jediným klíčem 22.

Příklad vkládání do B-stromu – vložení klíče 67 (pokrač.)

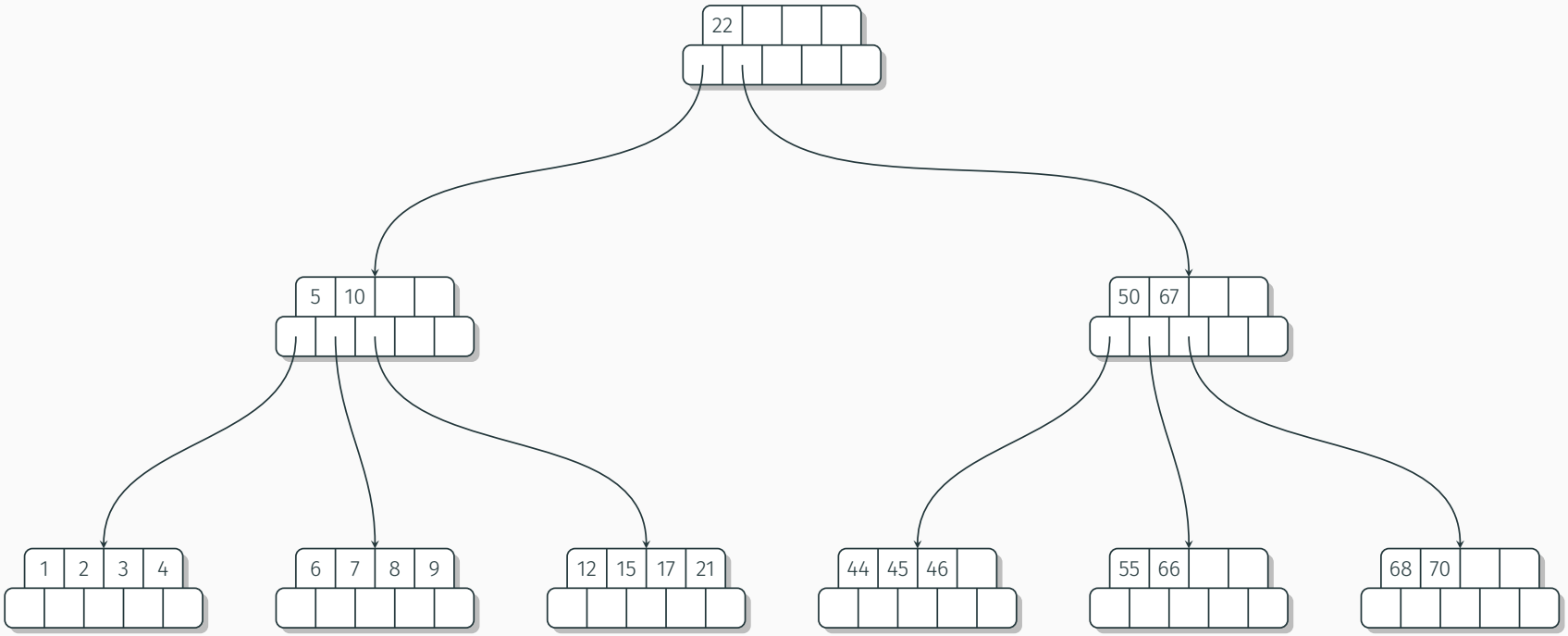
Poznámky

- V tomto okamžiku dosahuje faktor naplnění B-stromu minimální hodnoty $\approx$ **50%**. V B-stromu je maximum volného místa pro vkládání dalších klíčů.
- Minimální zaplnění B-stromu ale naopak způsobí, že odebráním libovolného klíče dojde ke slučování stránek[2] B-stromu, včetně zrušení kořene a následného snížení výšky B-stromu.

[2] Viz další část prezentace

Příklad vkládání do B-stromu – vložení dalších klíčů

Do stromu byly dále vloženy klíče 1, 4, 8, 9, 12, 15 a 46. Na pořadí vložení klíčů, v tomto případě, nezáleží.



B-stromy – smazání klíče x

1. Nejprve je nutné klíč x ve stromu najít.
2. Označme stránku s klíčem x jako P.
3. Mohou nastat dva případy:
   - stránka P je **vnitřní stránka** stromu nebo
   - stránka P je **listová stránka**.

B-stromy – smazání klíče x z vnitřní stránky P

1. Klíč x ve stránce P nahradíme k němu nejbližším větším klíčem y.
2. Klíč y se musí nacházet v podstromu s klíči většími než x a zároveň je z těchto klíčů nejmenší, musí se tedy nacházet v listové stránce.
3. Smazání klíče x z vnitřní stránky stromu jsme tak převedli na smazání klíče y z listové stránky stromu.

B-stromy – smazání klíče x z listové stránky P

1. Klíč x vymažeme ze stránky P.
2. Pokud stránka P i potom obsahuje aspoň n klíčů, je proces smazání ukončen.
3. Pokud P potom obsahuje jen $n - 1$ klíčů, musíme chybějící jeden klíč doplnit.
   - 3.1 Zjistíme počet klíčů v sourozenci stránky P. Sourozence označíme jako S. Společného rodiče stránek P a S označíme jako R.
   - 3.2 Pokud je v S je **více než** n klíčů, potom
     - 3.2.1 nejbližší větší klíč než x přesuneme z R do P a
     - 3.2.2 nejmenší z klíčů v S přesuneme do R.
   - 3.3 Ve stránce S je **přesně** n klíčů, potom

B-stromy – smazání klíče x z listové stránky P (pokrač.)

3.3.1 klíče ze stránky S přesuneme do stránky P a dostáváme jednu stránku s $2n - 1$ klíči.

3.3.2 Stránku S zrušíme.

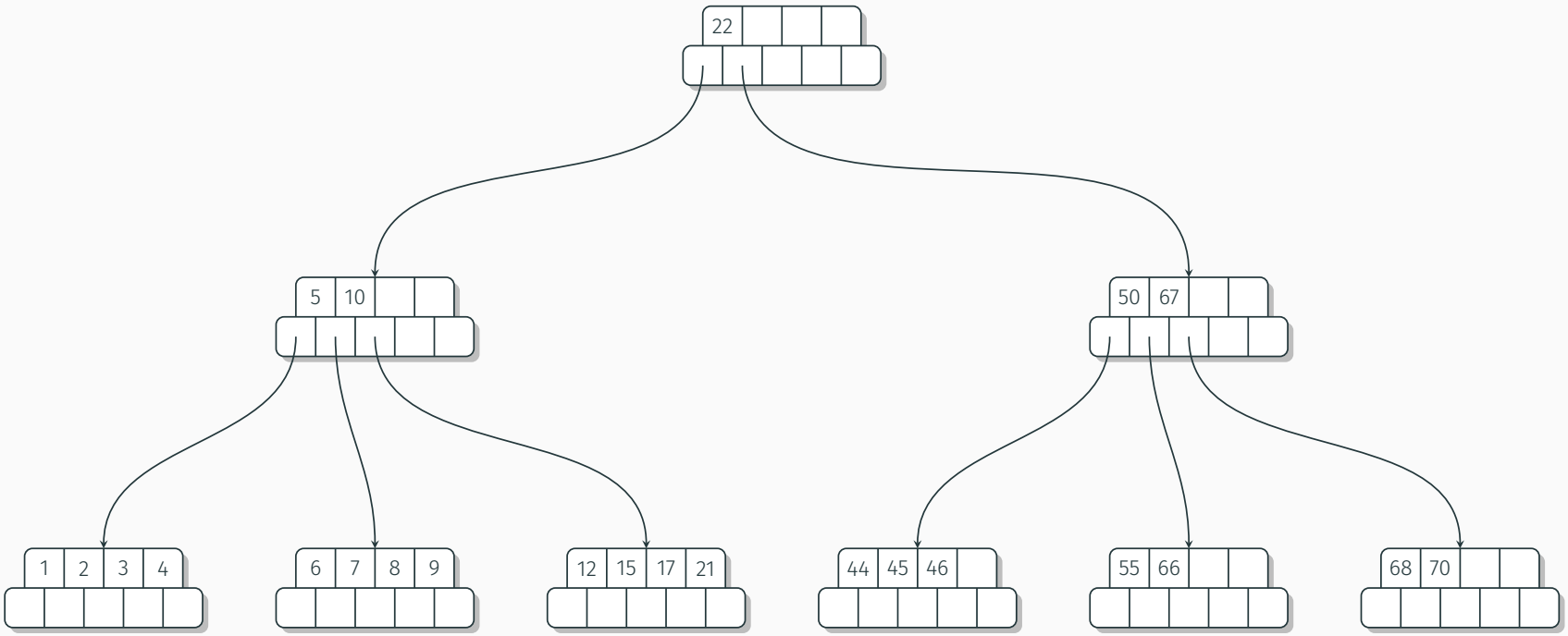
3.3.3 Ve stránce R je nyní jeden odkaz na stránku přebytečný. Nejbližší větší klíč než x přesuneme z R do stránky P, která nyní obsahuje přesně $2n$ klíčů.

B-stromy – smazání klíče x z listové stránky P (pokrač.)

Poznámky

- Obvykle volíme sourozence s většími klíči než je x, tedy sourozence „vpravo“ od P. V předchozím výkladu jsme předpokládali tuto volbu.
- Lze však volit i sourozence s menšími klíči, čili „vlevo“ od P. Další postup je v tomto případě zrcadlovým obrazem k sourozenci „vpravo“.
- Proces přesunu klíčů z S do P a následné zrušení stránky S se nazývá **slučování stránek**.
- Proces slučování stránek může pokračovat postupně až ke kořeni stromu a může vést i k zániku současného kořene stromu. Novým kořenem stromu pak bude stránka vzešlá z procesu slučování. B-strom tak sníží svoji výšku.

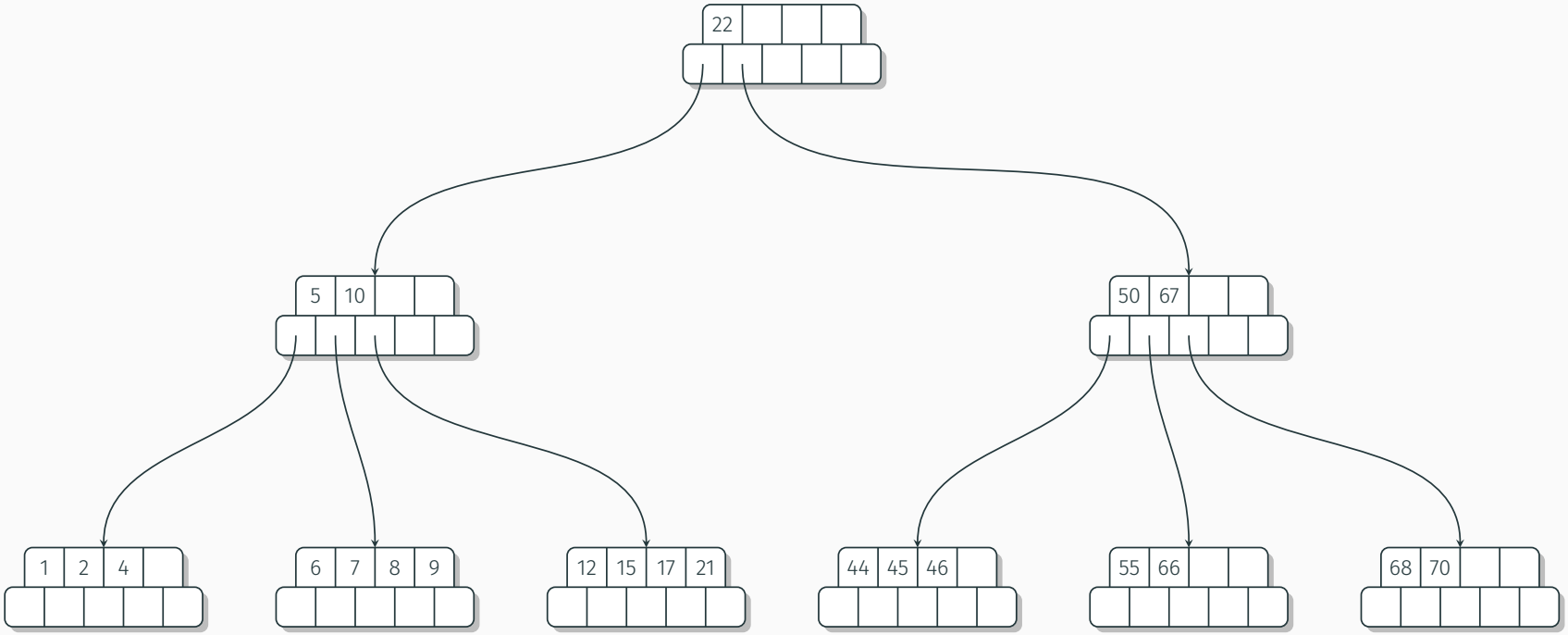
Příklad mazání v B-stromu – výchozí B-strom



V tomto stavu jsme B-strom zanechali na konci příkladu vkládání klíčů do B-stromu. Nyní budeme klíče postupně z B-stromu mazat.

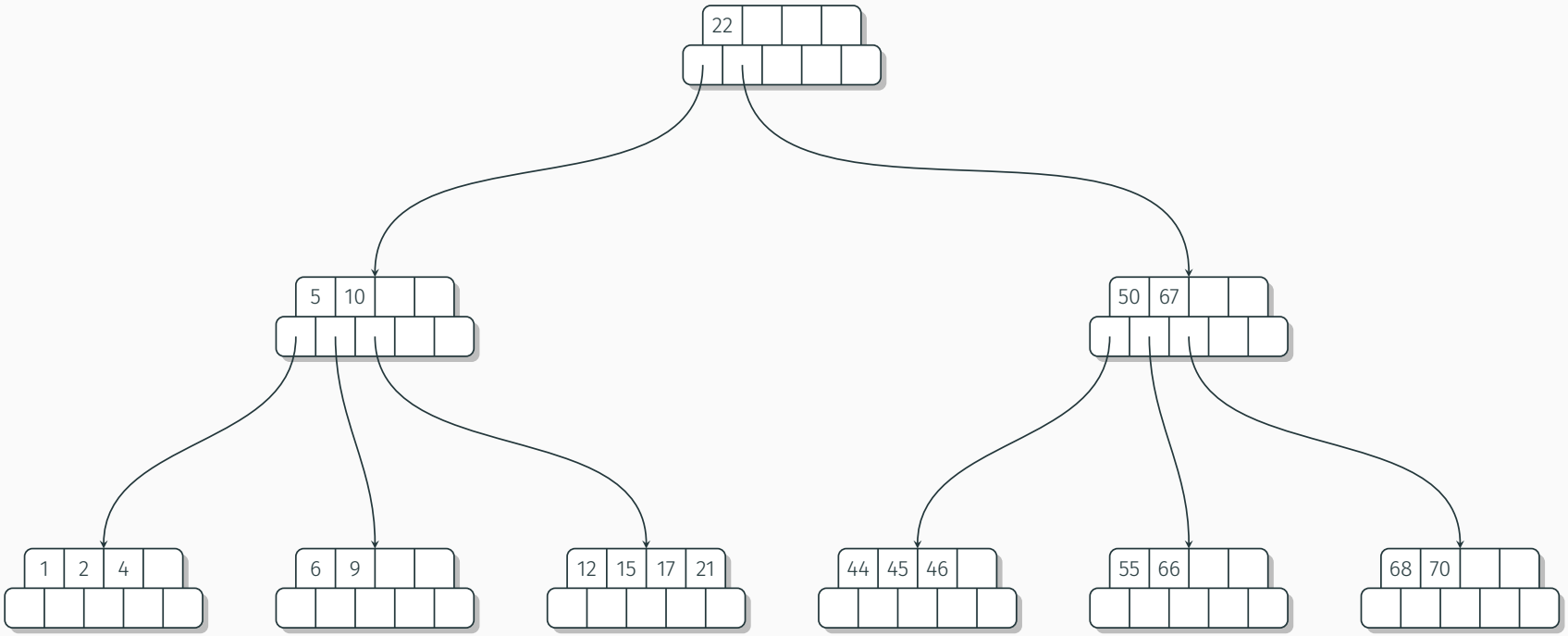
Příklad mazání v B-stromu – smazání klíče 3

Klíč 3 se nachází v listové stránce, kde je dostatečný počet klíčů na to, abychom klíč 3 mohli jednoduše smazat. Tím dostáváme následující B-strom.



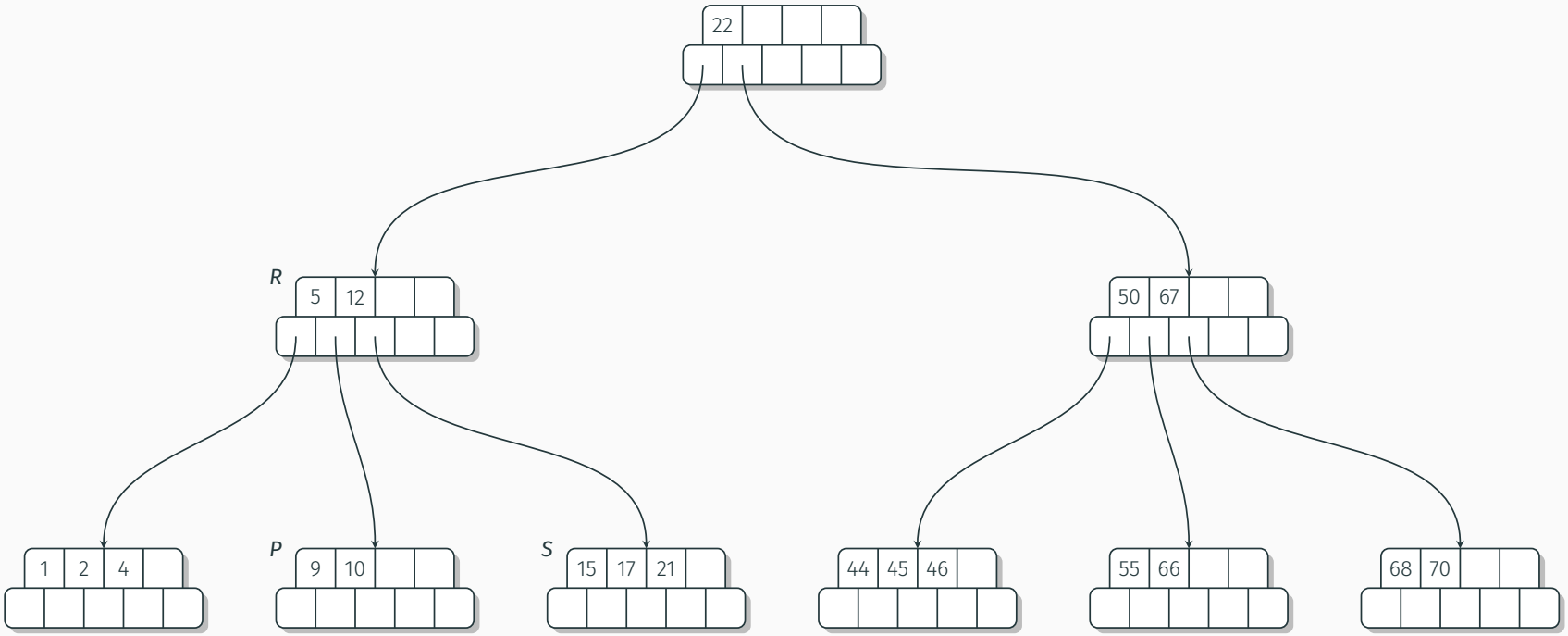
Příklad mazání v B-stromu – smazání klíčů 7 a 8

Stejným způsobem smažeme klíče 7, 8 a dostáváme



Příklad mazání v B-stromu – smazání klíče 6

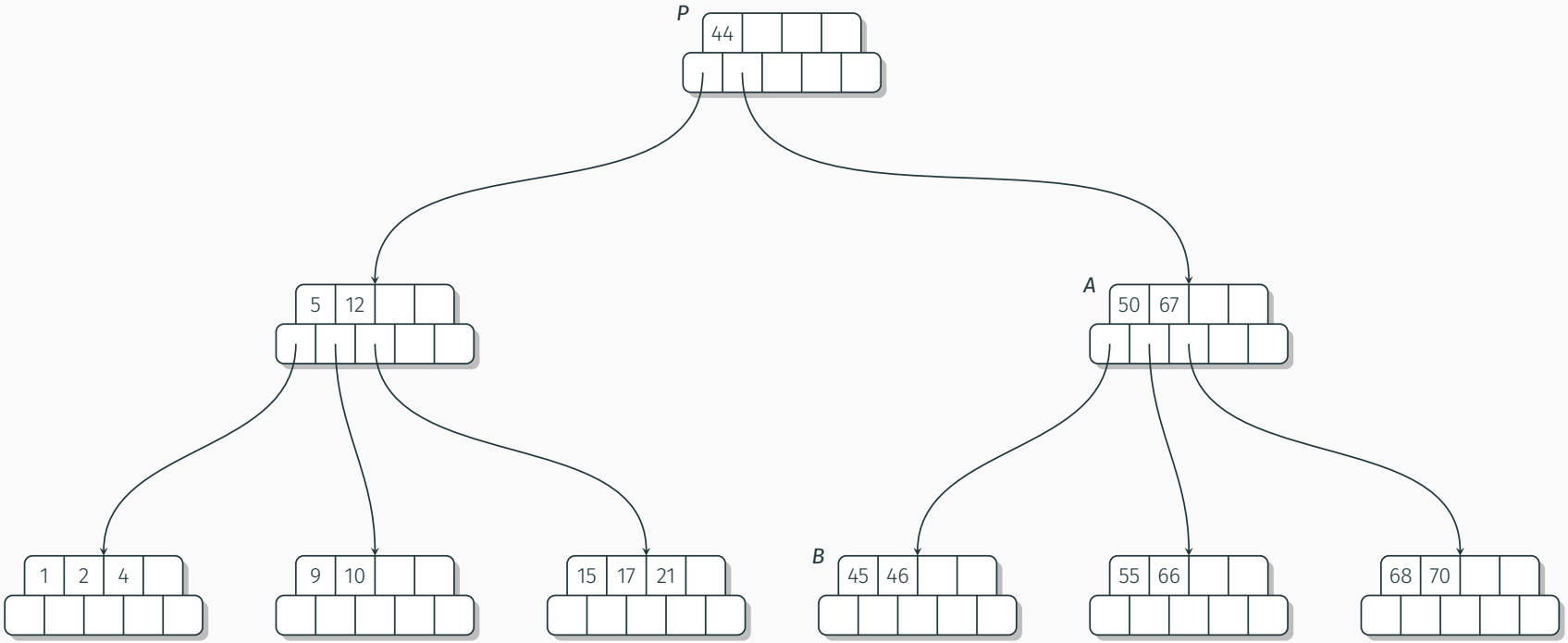
1. Po smazání klíče 6 obsahuje stránka P $n - 1 = 1$ klíč, číslo 9.
2. Sourozenec S obsahuje více než n klíčů.
3. Nejbližší větší klíč než 6, tj. 10, přesuneme z R do P.
4. Nejmenší klíč z S, tj. 12, přesuneme do R.



Příklad mazání v B-stromu – smazání klíče 22

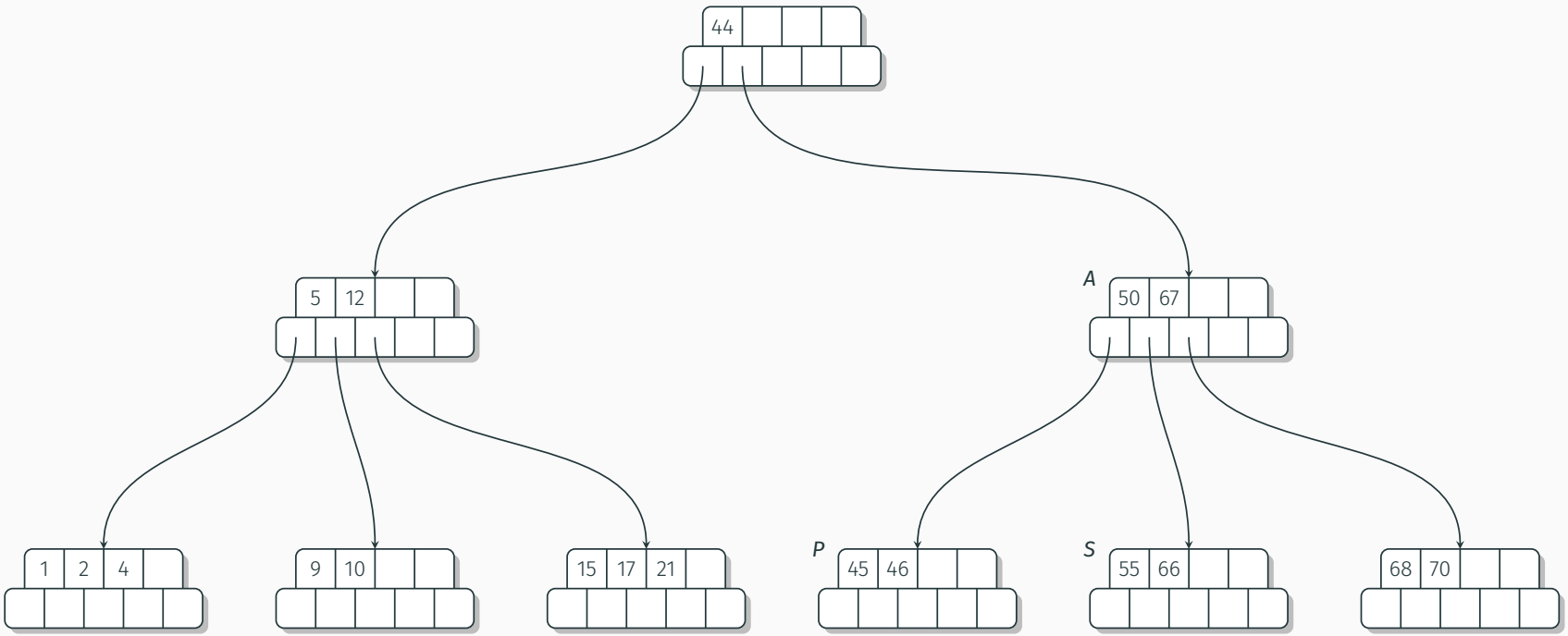
1. Klíč 22 se nachází ve vnitřní stránce *P*, viz následující obrázek.
2. Nahradíme jej nejbližším větším klíčem – větší klíče než 22 jsou v podstromu s kořenem ve stránce *A*. Odtud pokračujeme do nejlevější listové stránky, v našem případě do *B*.
3. Vybereme nejmenší klíč v *B*, tj. 44.
4. Tím jsme převedli smazání klíče 22 na smazání klíče 44.
5. Po provedení všech operací, které odpovídají smazání 44 (v tomto případě se jedná jen o smazání 44 ze stránky *B*), nahradíme klíčem 44 klíč 22.

Příklad mazání v B-stromu – smazání klíče 22 (pokrač.)



Příklad mazání v B-stromu – smazání klíče 46, fáze I

Stav B-stromu před zahájením mazání

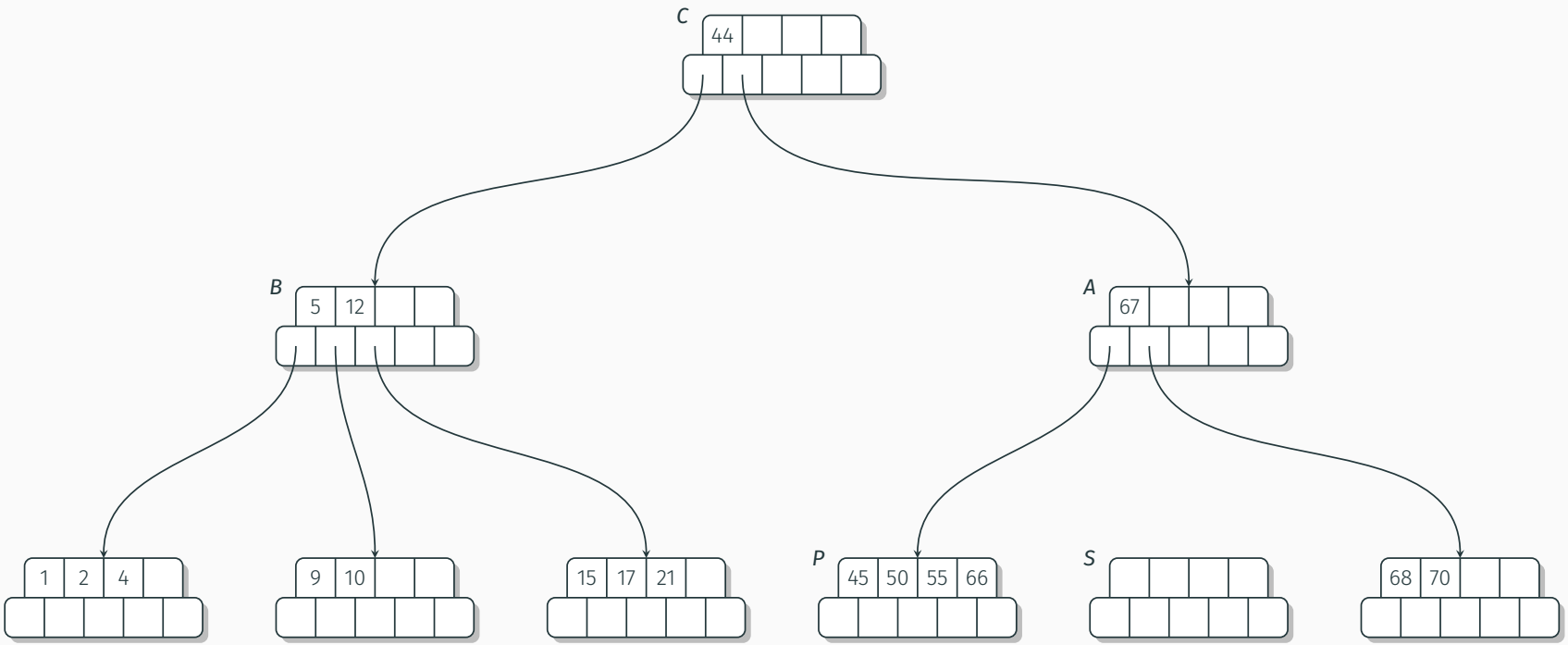


Příklad mazání v B-stromu – smazání klíče 46, fáze I (pokrač.)

1. Po smazání klíče 46 obsahuje stránka P $n - 1$ klíčů, tj. pouze klíč 45.
2. Sourozenec S obsahuje přesně než n klíčů, musíme slučovat stránky.
3. Všechny klíče z S přesuneme do P.
4. Stránka P je prvním potomkem stránky A, přesuneme tedy i první klíč z A do P.

Příklad mazání v B-stromu – smazání klíče 46, fáze I (pokrač.)

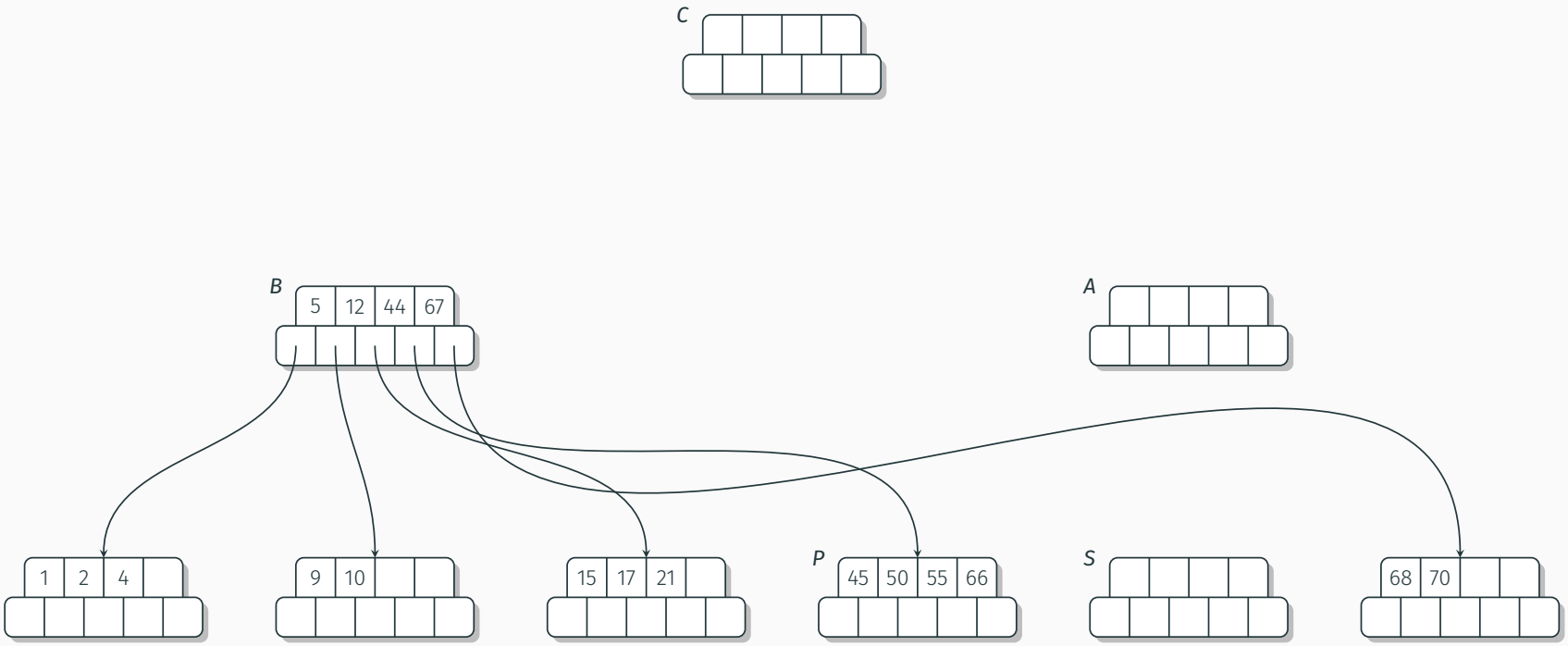
Výsledek fáze 1



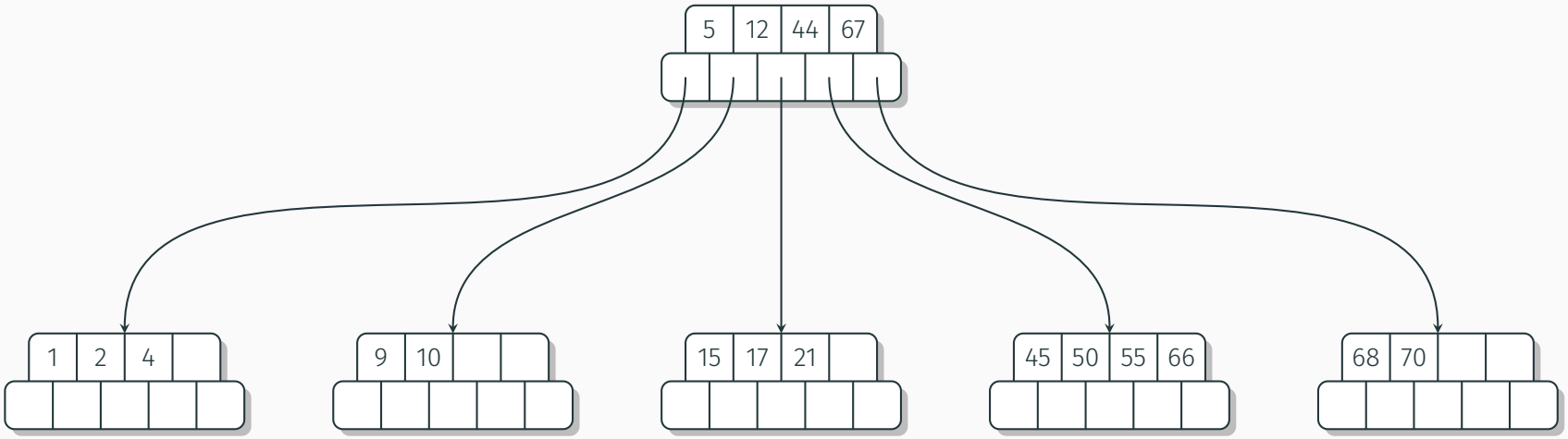
Příklad mazání v B-stromu – smazání klíče 46, fáze II

1. Ve stránce A zůstalo jen $n - 1$ klíčů, což odporuje definici B-stromu.
2. Sourozenec B obsahuje přesně n klíčů, musíme i na této úrovni provést slučování stránek.
3. Přesuneme klíč 67 ze stránky A do stránky B.
4. A stejně tak do B přesuneme i jeden klíč z rodičovské stránky C.
5. Tím došlo ke zrušení kořenové stránky a ke snížení výšky B-stromu.

Příklad mazání v B-stromu – smazání klíče 46, fáze II (pokrač.)

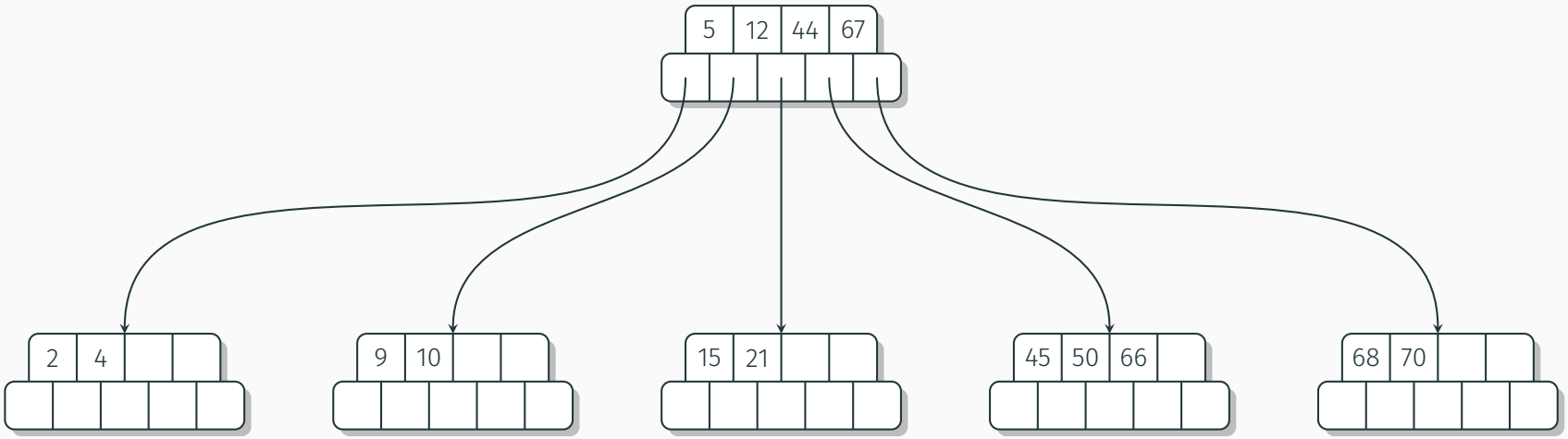


Příklad mazání v B-stromu – smazání klíče 46, výsledek



Příklad mazání v B-stromu – smazání klíčů 1, 17 a 55

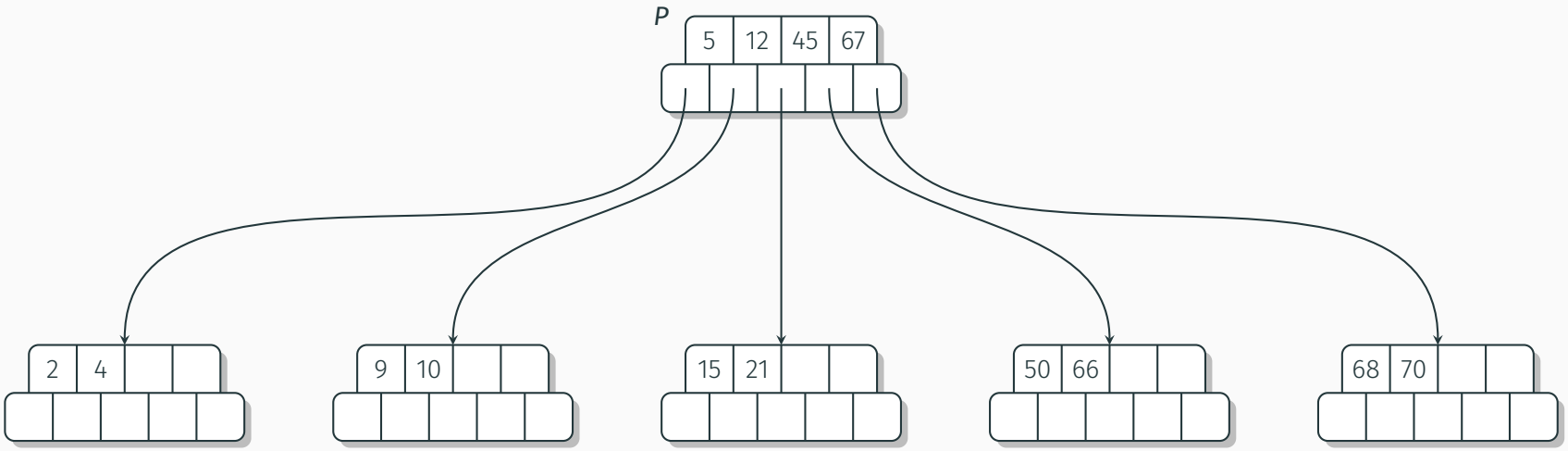
Klíče 1, 17 a 55 se nachází v listových stránkách, kde je dostatečný počet klíčů na to, abychom je mohli jednoduše smazat.



Příklad mazání v B-stromu – smazání klíče 44

1. Klíč 44 se nachází ve vnitřní stránce.
2. Nahradíme jej nejbližším větším klíčem, tj. klíčem 45.

Výsledný strom

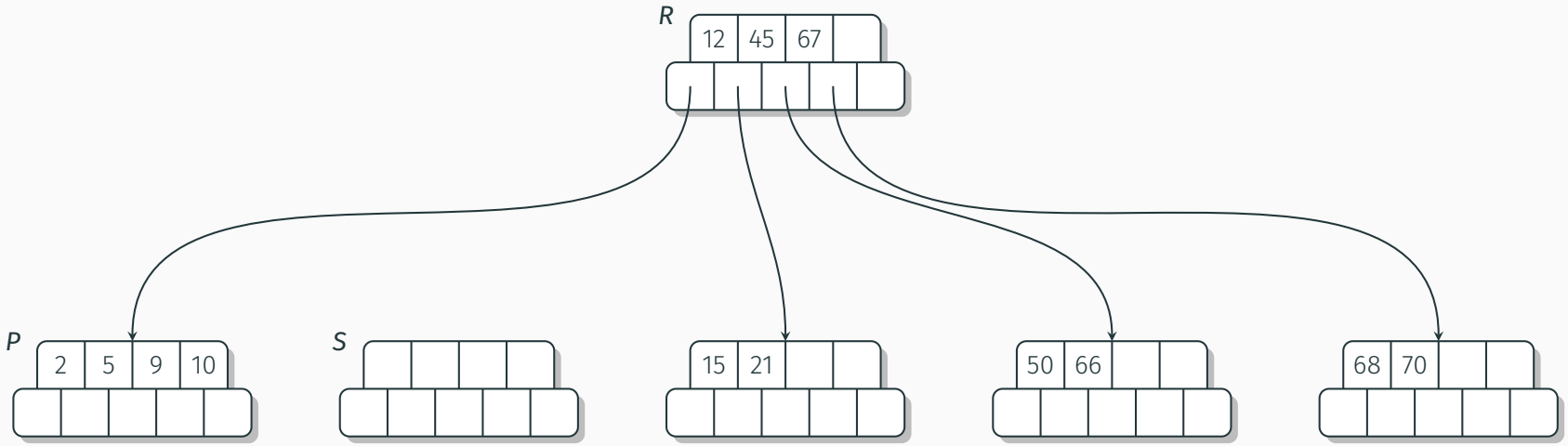


B-strom se nyní nachází ve stavu, kdy listové stránky jsou naplněny na minimální přípustnou úroveň. Smazání jakéhokoliv klíče způsobí sloučení stránek.

Příklad mazání v B-stromu – smazání klíče 4

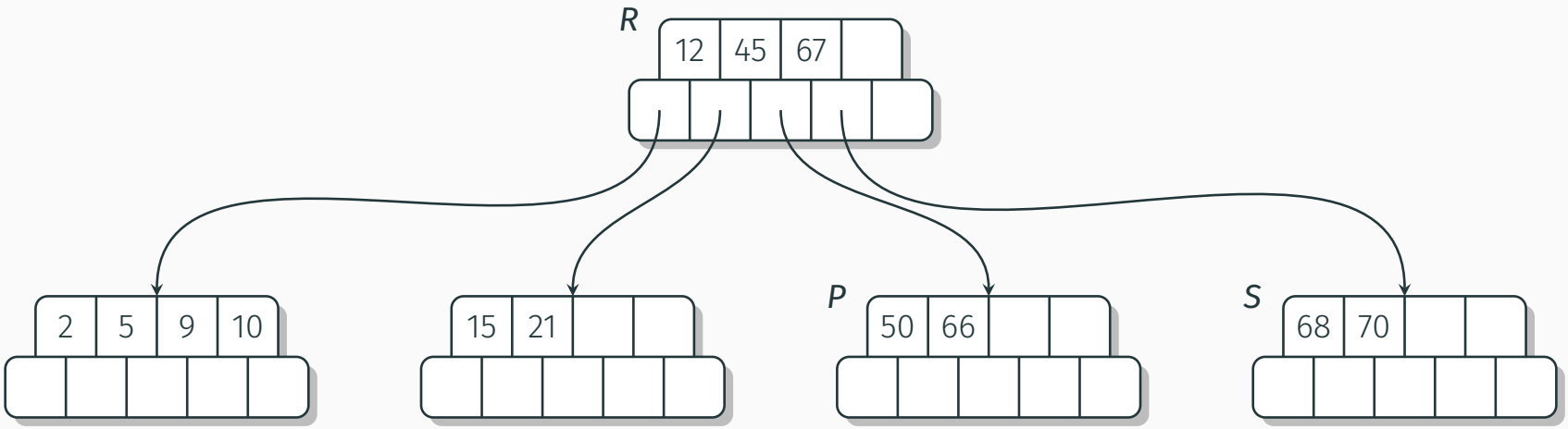
1. Po smazání klíče 4 obsahuje stránka P pouze $n - 1$ klíčů.
2. Sourozenec S obsahuje přesně n klíčů.
3. Klíče z S přesuneme do P.
4. Do P přesuneme i klíč 5 z rodičovské stránky R, protože v R by jinak jeden odkaz na potomky přebýval.

Výsledný strom



Příklad mazání v B-stromu – smazání klíče 45

Strom před smazáním klíče 45



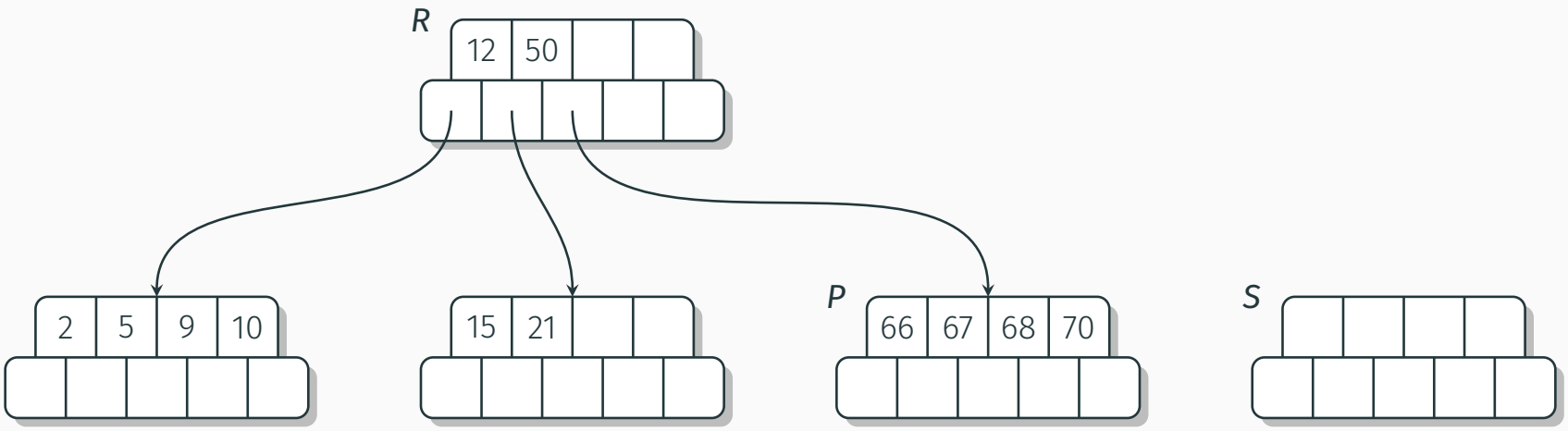
1. Klíč 45 se nachází ve vnitřní stránce R.
2. Nahradíme jej nejbližším větším klíčem, tj. klíčem 50.
3. Tím jsme převedli smazání klíče 45 na smazání klíče 50.

Příklad mazání v B-stromu – smazání klíče 45 (pokrač.)

4. Po smazání klíče 50 obsahuje stránka P pouze $n - 1$ klíčů.
5. Sourozenec S obsahuje přesně n klíčů.
6. Klíče z S přesuneme do P.
7. Do P přesuneme i nejbližší větší klíč než 45, tj. klíč 57, z rodičovské stránky R, protože v R by jinak jeden odkaz na potomky přebýval.

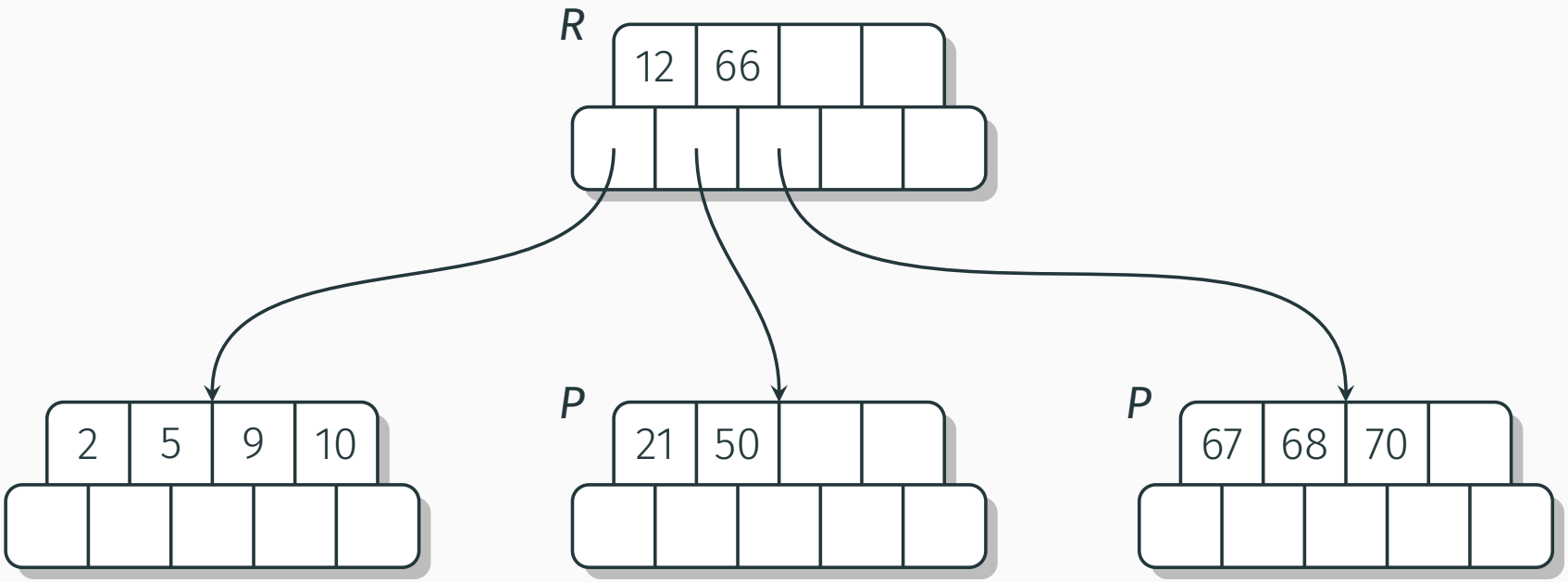
Příklad mazání v B-stromu – smazání klíče 45 (pokrač.)

Výsledný strom



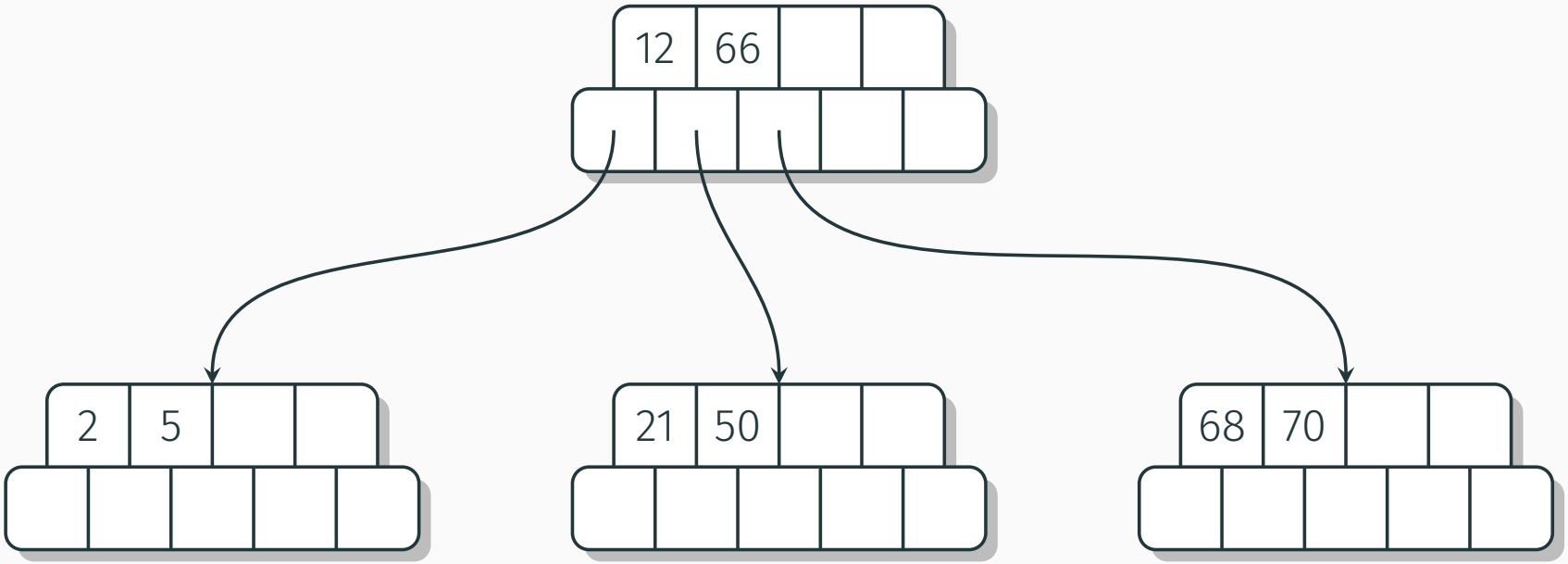
Příklad mazání v B-stromu – smazání klíče 15

Po smazání 15 zůstalo ve stránce P pouze $n - 1$ klíčů. Musíme tedy přesunout přes stránku R jeden klíč ze stránky S.



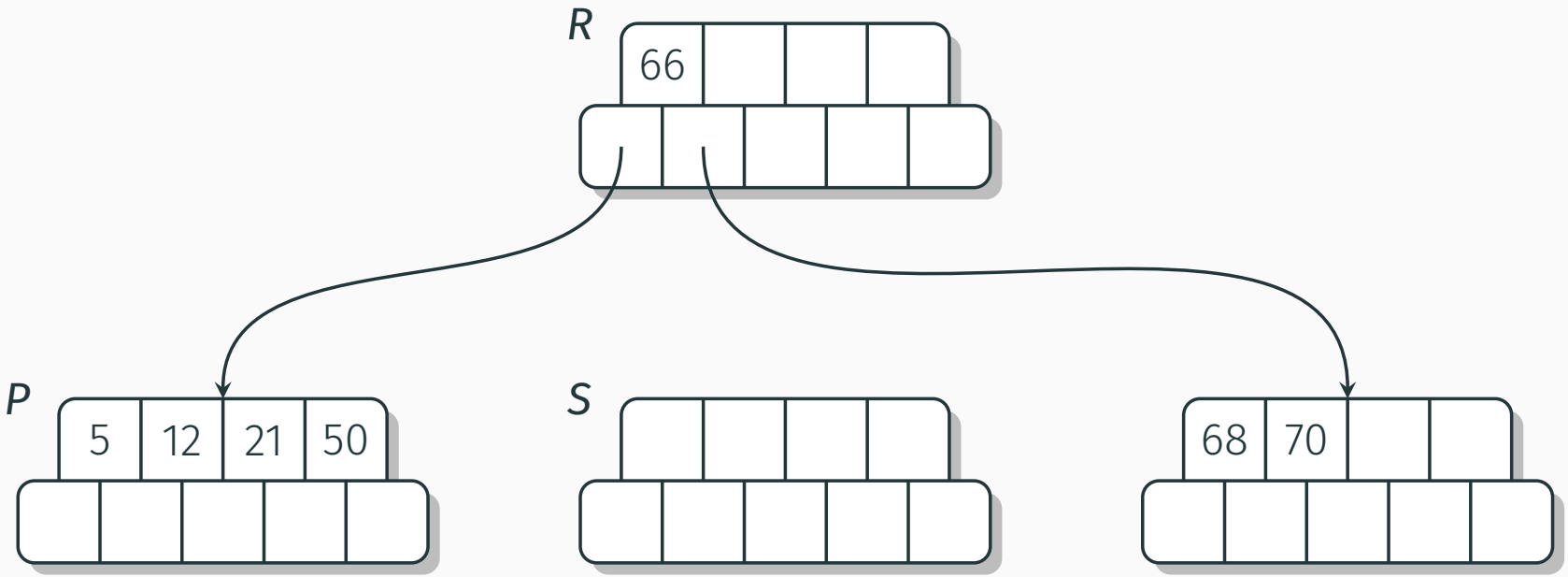
Příklad mazání v B-stromu – smazání klíčů 9, 10 a 67

Smazání klíčů 9, 10 a 67 je velice jednoduché.

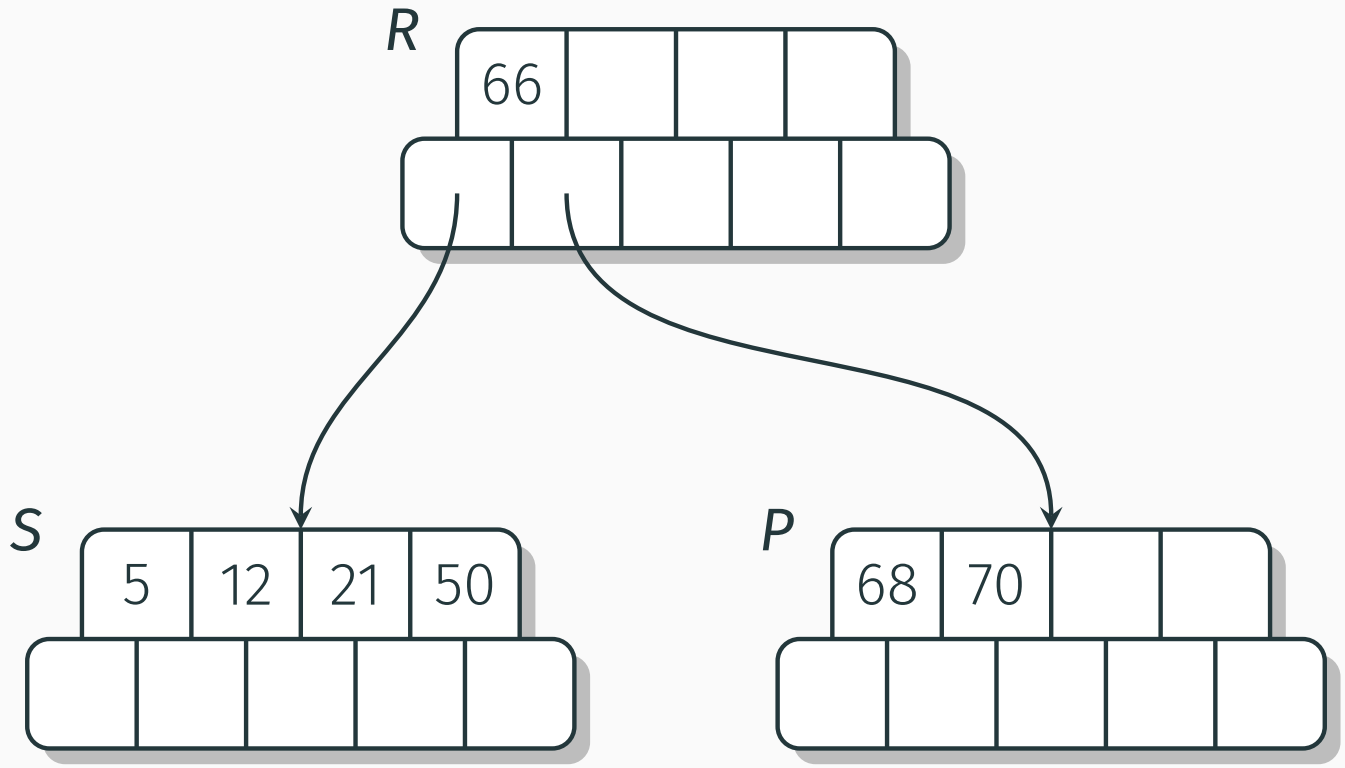


Příklad mazání v B-stromu – smazání klíče 2

Po smazání 2 zůstalo ve stránce P pouze $n - 1$ klíčů. Sourozenec S obsahuje n klíčů, dojde tedy ke slučování stránek.

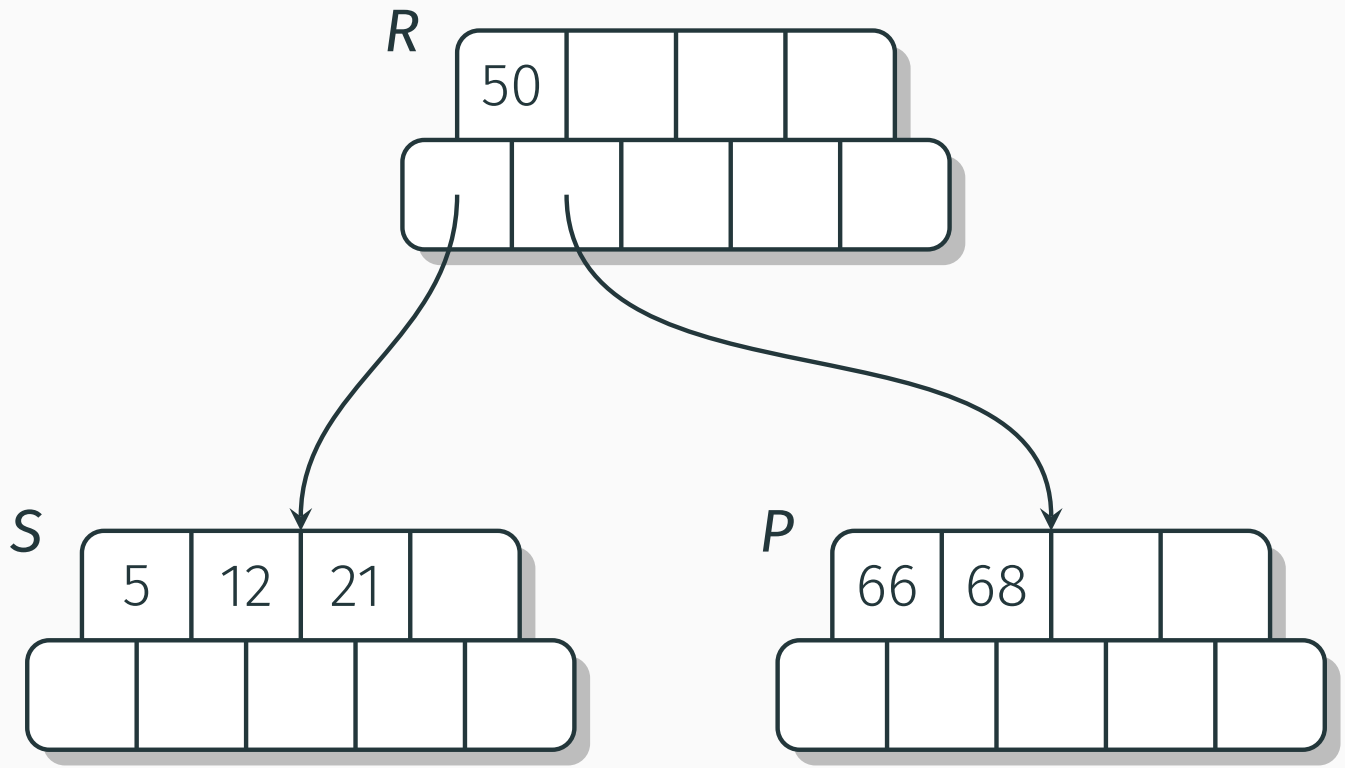


Příklad mazání v B-stromu – smazání klíče 70



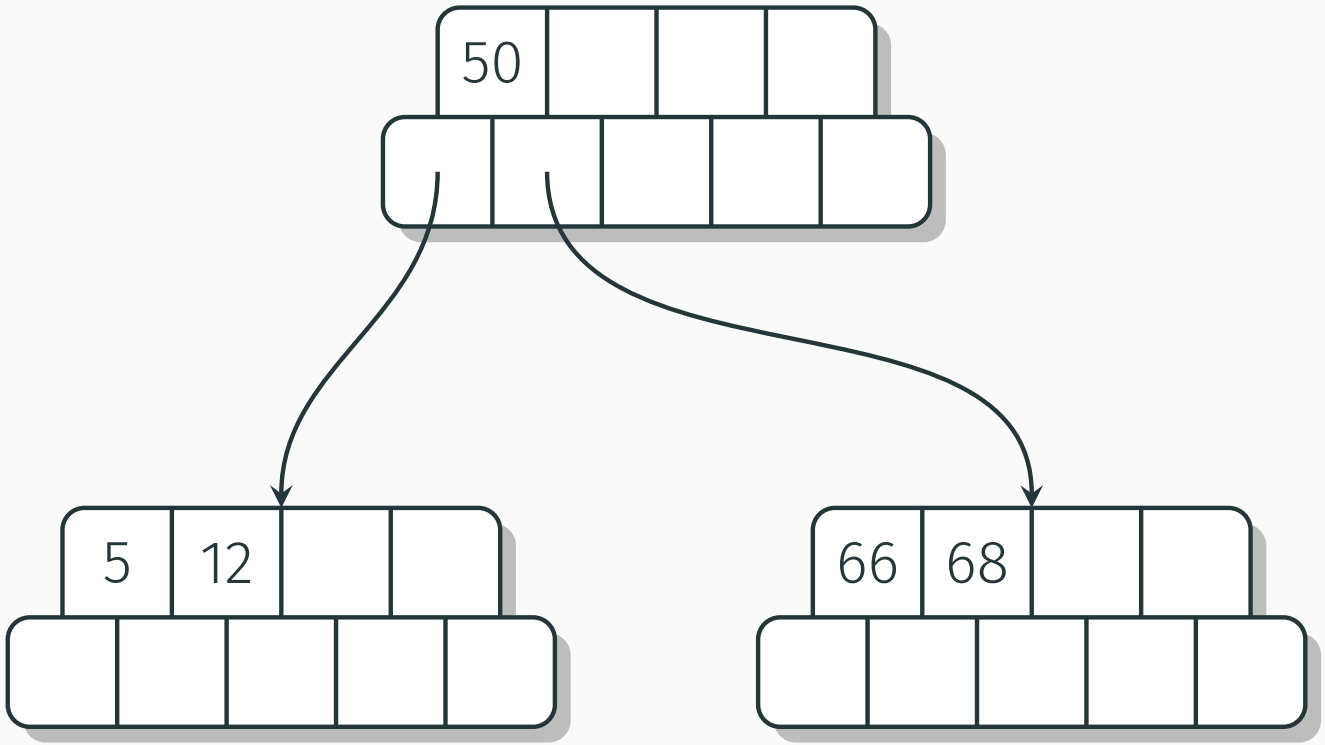
Příklad mazání v B-stromu – smazání klíče 70 (pokrač.)

Po smazání 70 zůstalo ve stránce P pouze $n - 1$ klíčů.
Sourozenec S obsahuje více n klíčů, dojde tedy k přesunu 66 z R do P a nejbližšího menšího klíče z S do R.



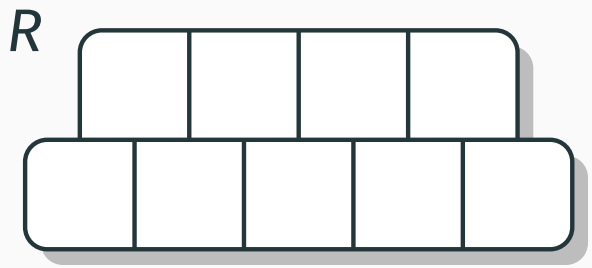
Příklad mazání v B-stromu – smazání klíče 21

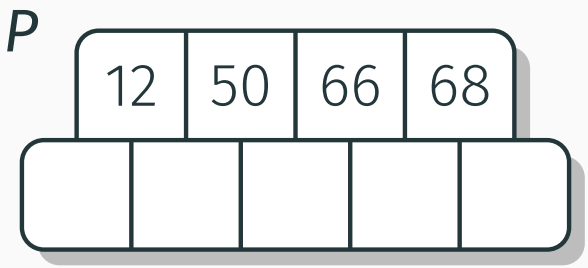
Smazání klíče 21 je velice jednoduché.

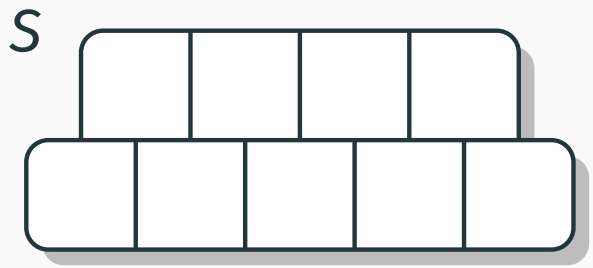


Příklad mazání v B-stromu – smazání klíče 5

Smazání klíče 5 je zřejmé.

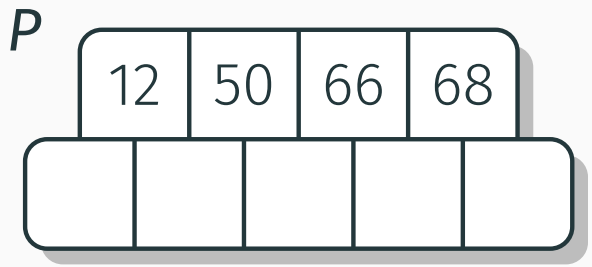






Příklad mazání v B-stromu – smazání klíče 5 (pokrač.)

Stránka *P* se stala novým kořenem, a současně jedinou stránkou, B-stromu.



Smazání klíčů 12, 50, 66 a 68 je už triviální záležitostí.

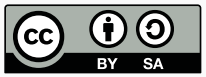
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Dynamické programování

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



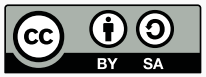
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Hladové algoritmy

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



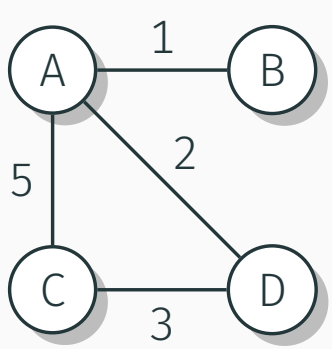
Hladové algoritmy

Minimální kostra grafu

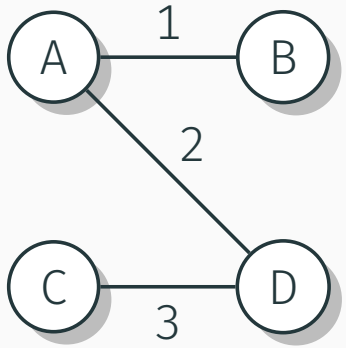
Minimální kostra grafu

Minimální kostra grafu – příklad

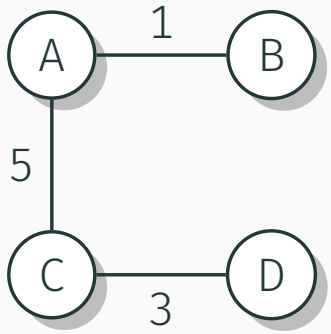
Graf G má celkem 3 kostry



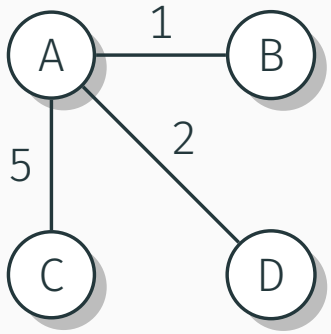
Graf G



$w(K_1) = 6$



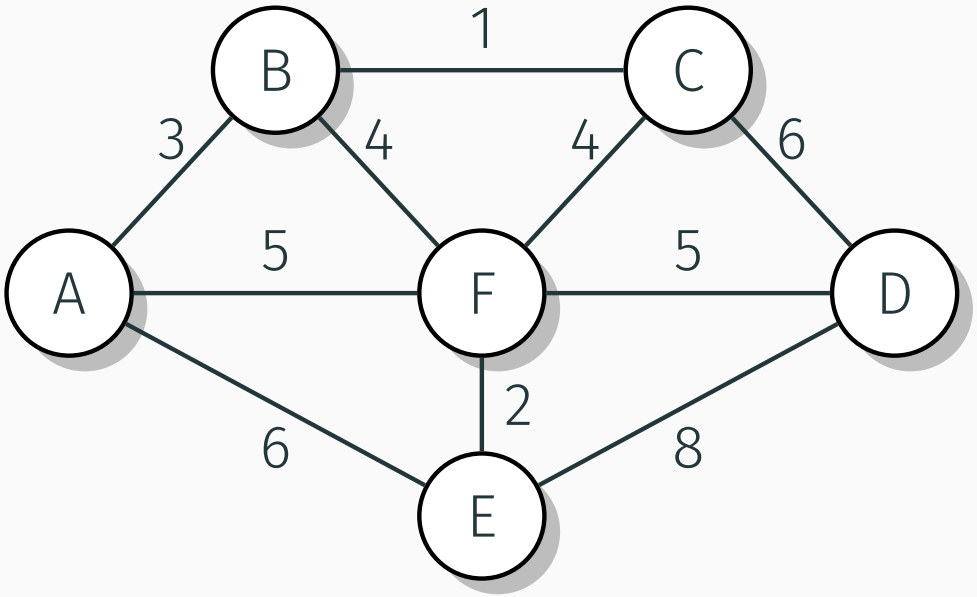
$w(K_2) = 9$



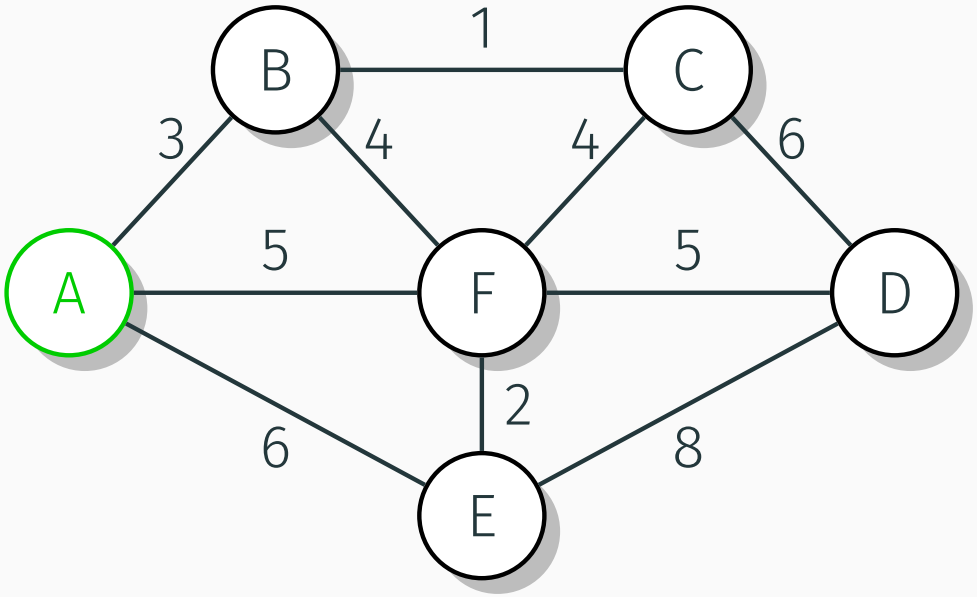
$w(K_3) = 8$

Minimální kostrou je kostra K_1.

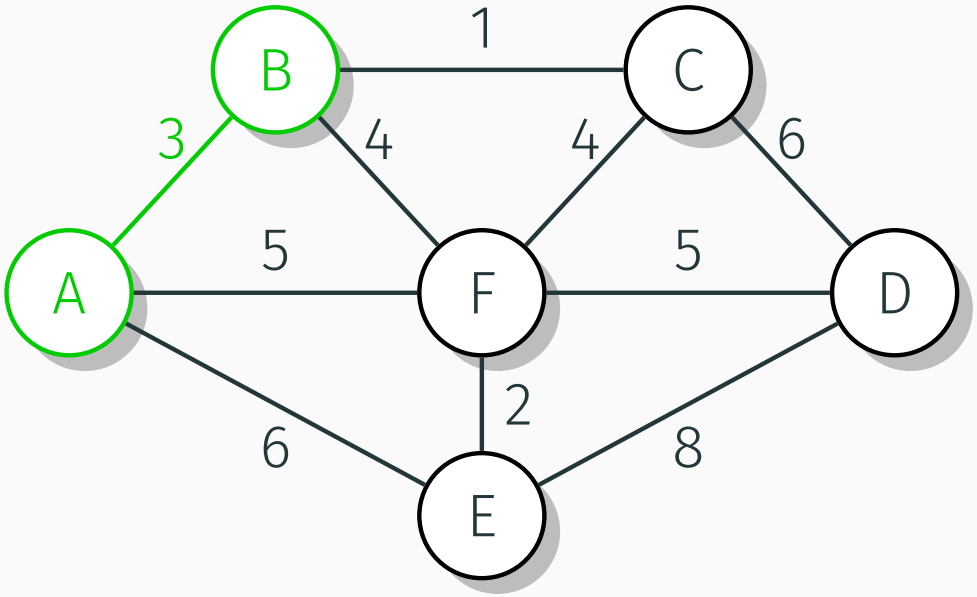
Minimální kostra grafu



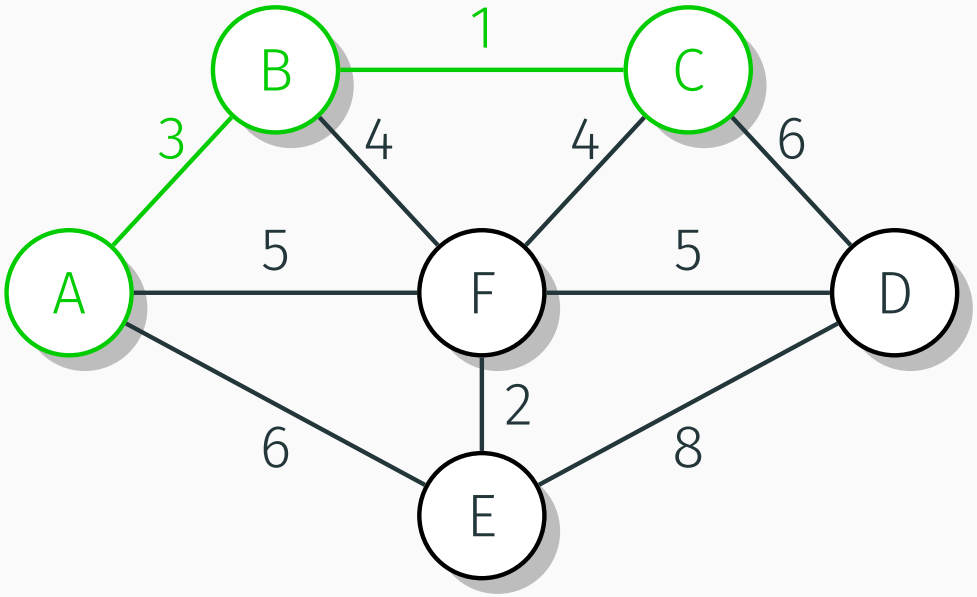
Minimální kostra grafu (pokrač.)



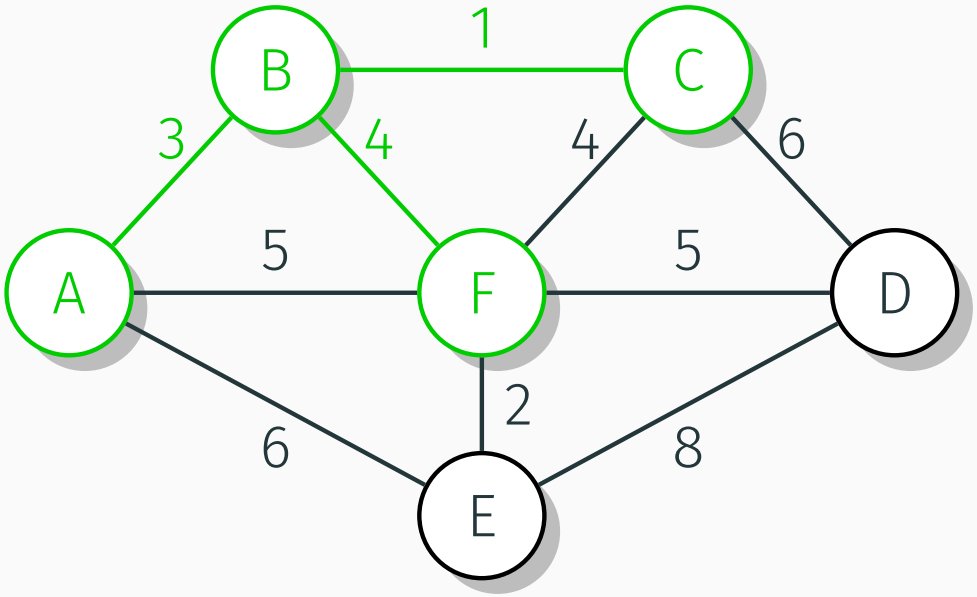
Minimální kostra grafu (pokrač.)



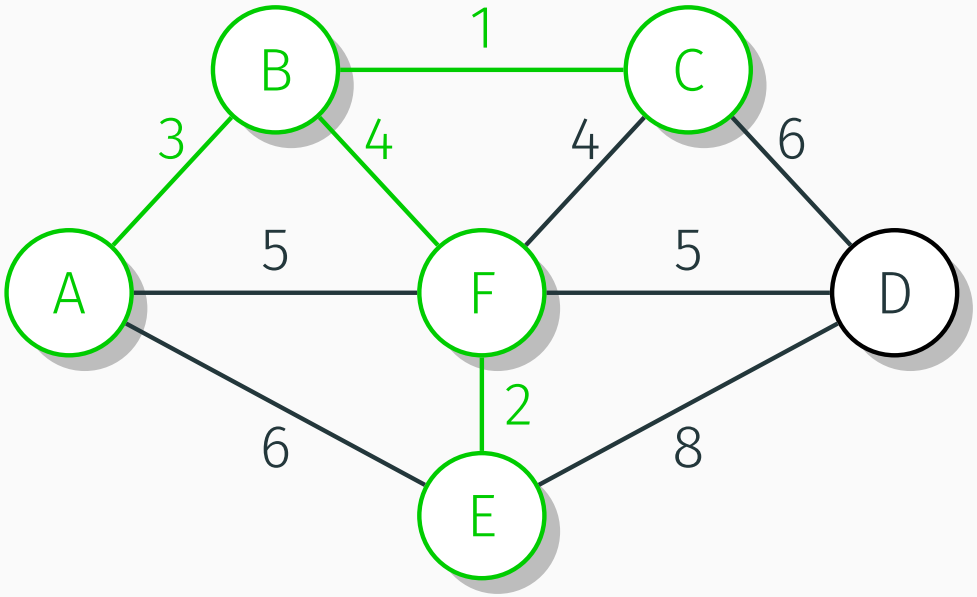
Minimální kostra grafu (pokrač.)



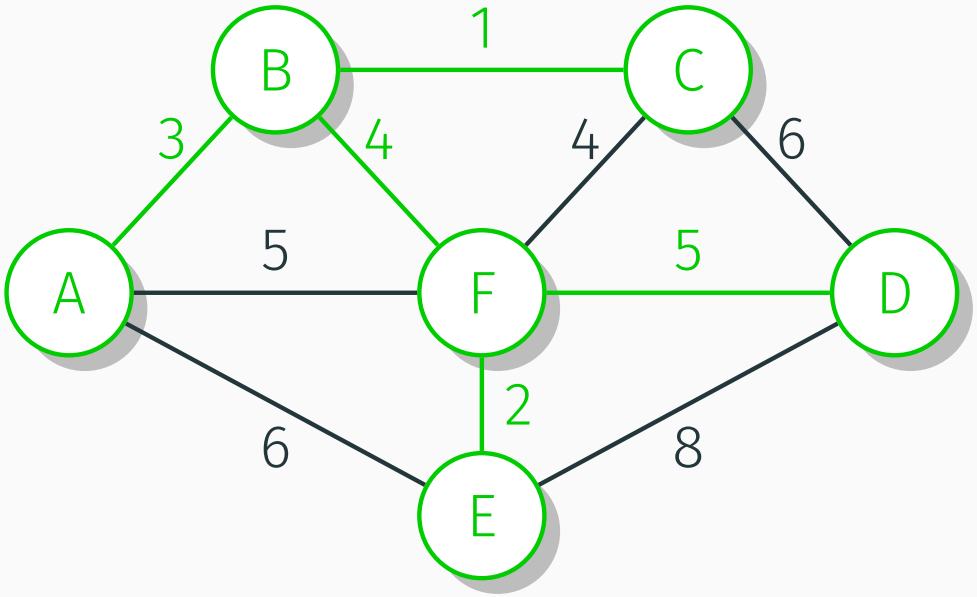
Minimální kostra grafu (pokrač.)



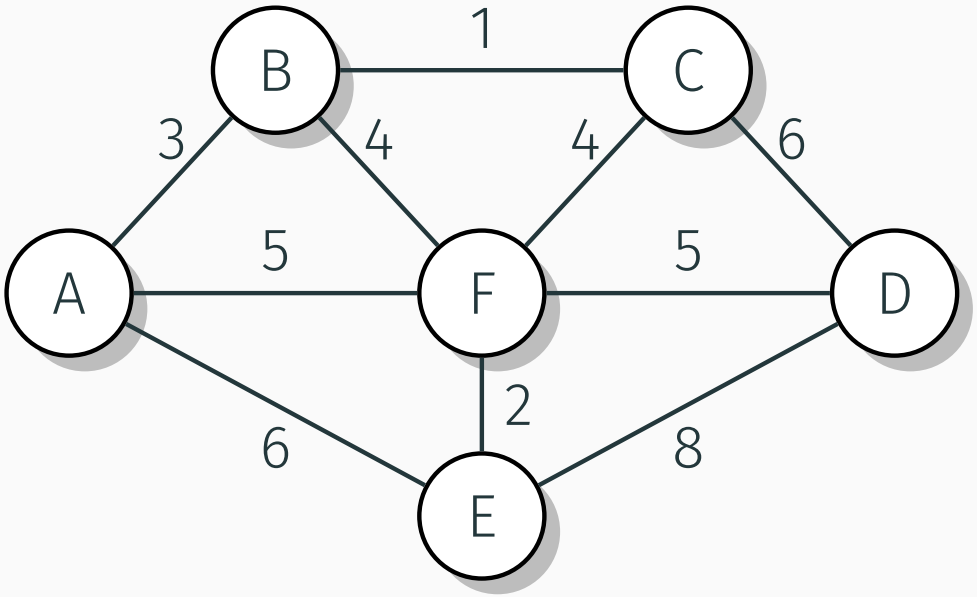
Minimální kostra grafu (pokrač.)



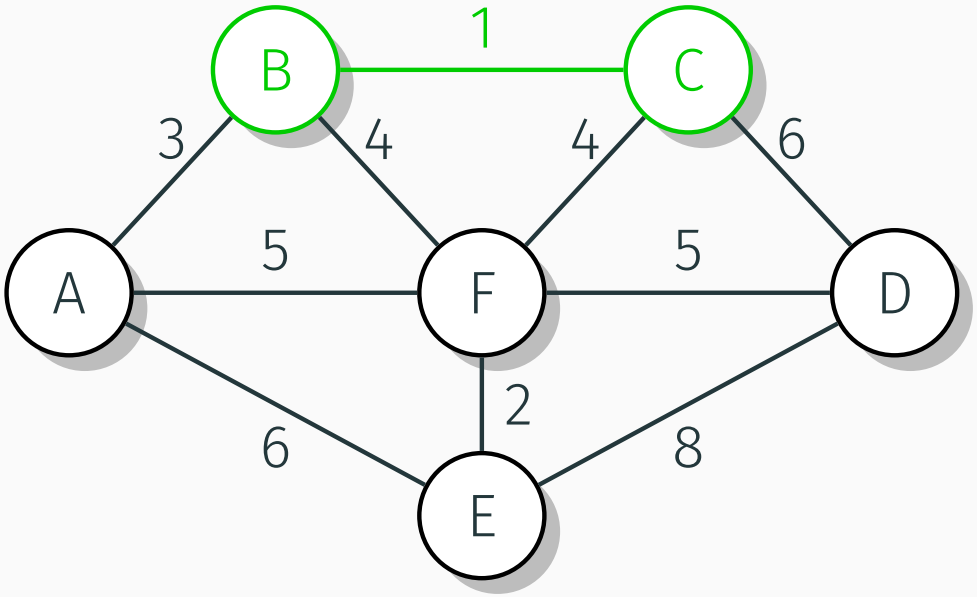
Minimální kostra grafu (pokrač.)



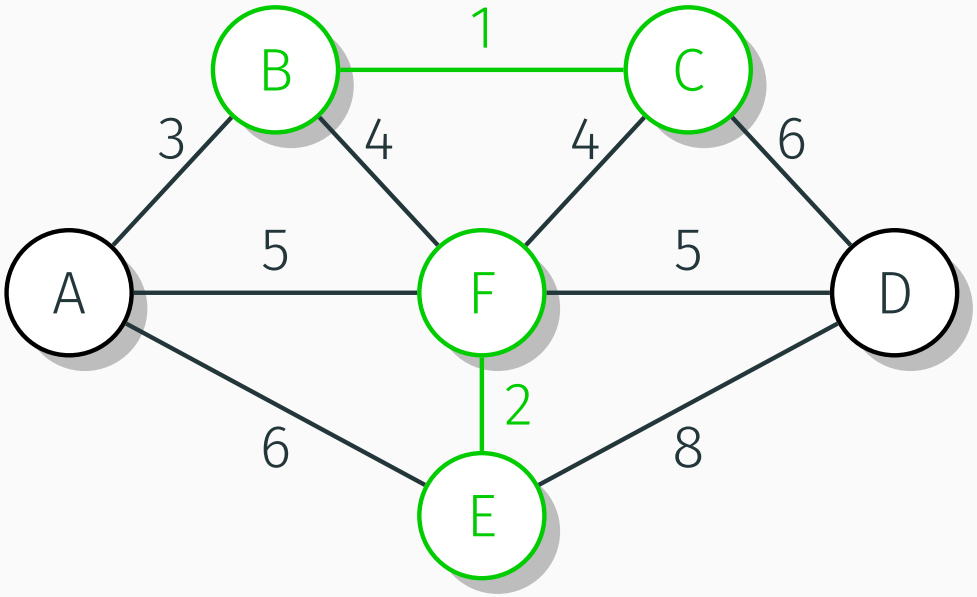
Minimální kostra grafu



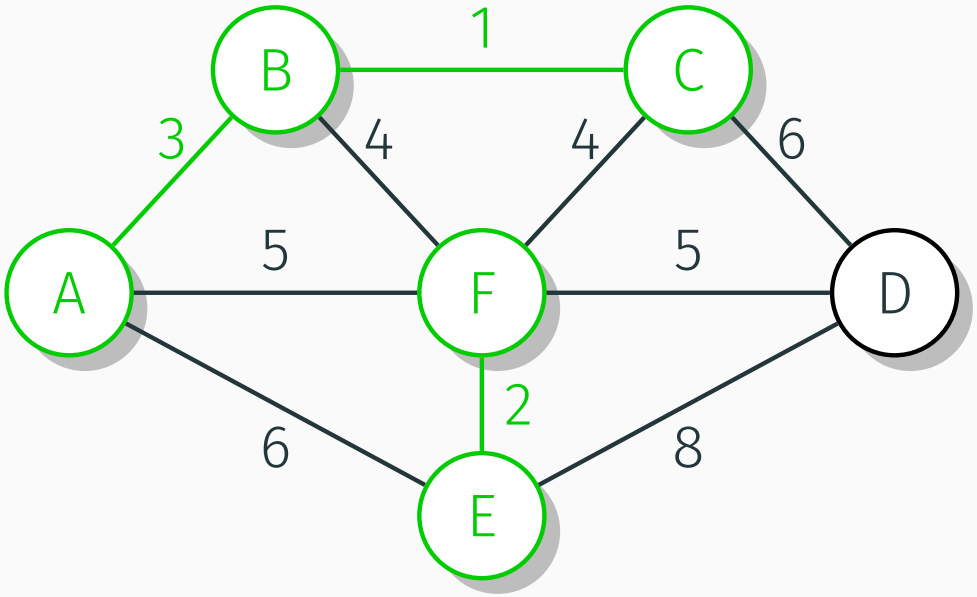
Minimální kostra grafu (pokrač.)



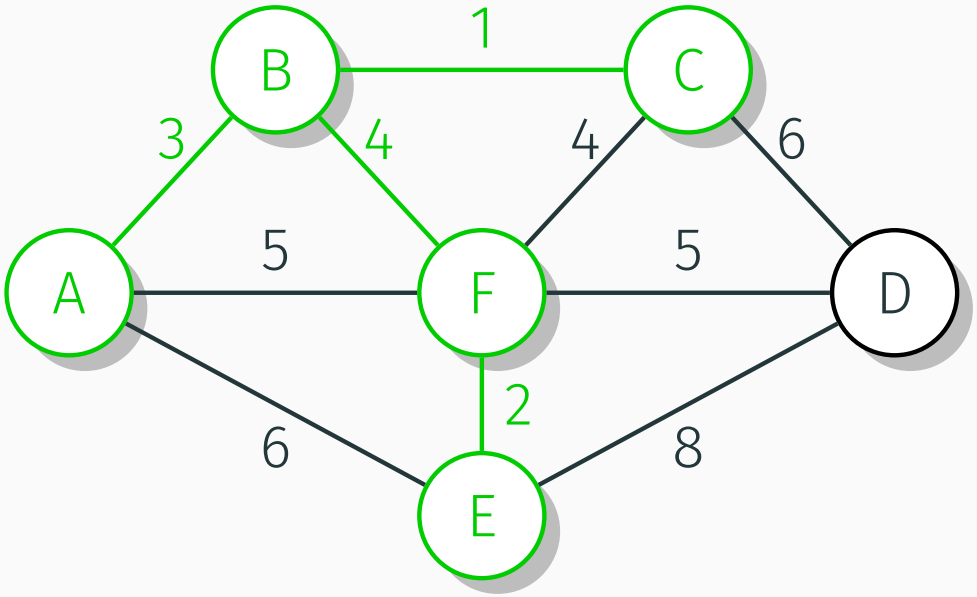
Minimální kostra grafu (pokrač.)



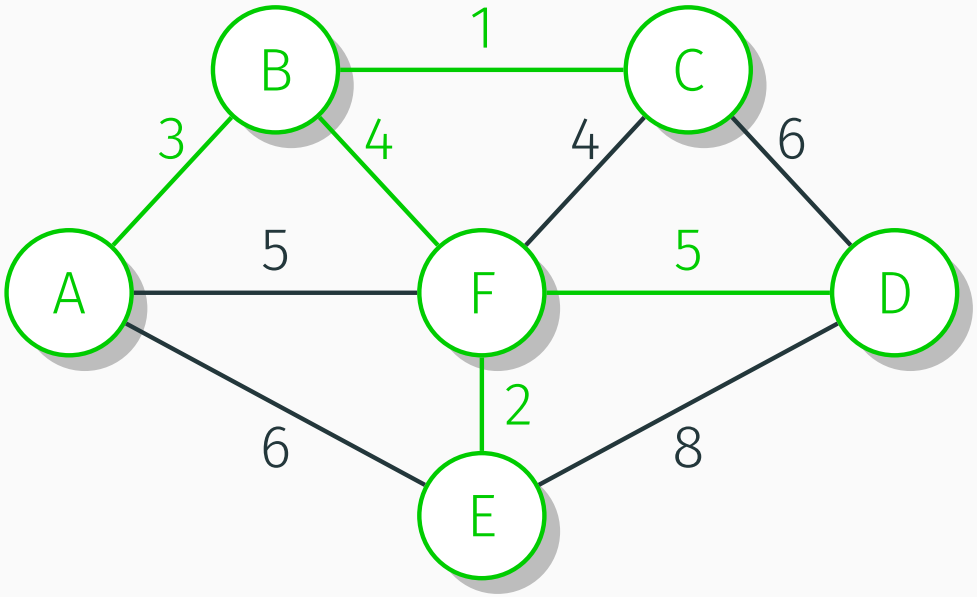
Minimální kostra grafu (pokrač.)



Minimální kostra grafu (pokrač.)



Minimální kostra grafu (pokrač.)



Hladové algoritmy

Dijkstrův algoritmus

Hladové algoritmy

Huffmanův kód

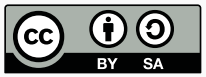
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Strategie řešení iterativním zlepšováním

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



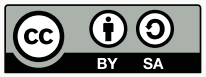
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Meze možností algoritmického řešení problémů. P, NP a NP-úplné problémy.

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



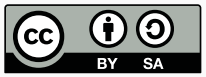
Děkuji za pozornost

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Zdolávání mezí možností algoritmického řešení problémů

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Děkuji za pozornost

Literatura

1. **Železniční mapy ČR** [online]. Praha: SŽDC, 2019 [cit. 2019-11-15]. Dostupné z: *http://provoz.szdc.cz/portal/Show.aspx?path=/Data/Mapy/kjr.pdf*.
2. LEVITIN, Anany. **Introduction to the Design and Analysis of Algorithms**. 3rd ed. Boston: Pearson, 2012. ISBN 978-0-13-231681-1.
3. CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L.; STEIN, Clifford. **Introduction to algorithms**. Fourth edition. Cambridge, Massachusetts: The MIT Press, [2022]. ISBN 978-026-2046-305.

Literatura (pokrač.)

4. WRÓBLEWSKI, Piotr. **Algoritmy**. 1. vyd. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.
5. BEČVÁŘ, Jindřich. **Lineární algebra**. Vyd. 4. Praha: Matfyzpress, 2010. ISBN 978-80-7378-135-4.
6. ROHN, Jiří. **Lineární algebra a optimalizace**. Praha: Karolinum, 2004. ISBN 80-246-0932-0.
7. WIRTH, Niklaus. **Algoritmy a štruktúry údajov**. 1. vyd. Bratislava: Alfa, 1988. ISBN 063-030-87.

Literatura (pokrač.)

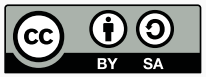
8. BAYER, Rudolf; MCCREIGHT, Edward Meyers. Organization and maintenance of large ordered indexes. **Acta Informatica**. 1972, roč. 1, č. 3, s. 173–189. ISSN 1432-0525. Dostupné z DOI: *10.1007/BF00288683*.

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Přílohy

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Přílohy

Zásobník

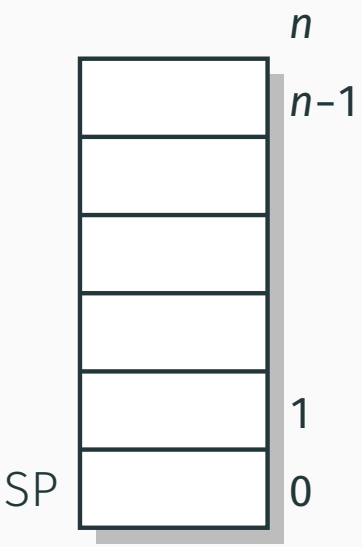
Zásobník – možnosti implementace

Pomocí pole

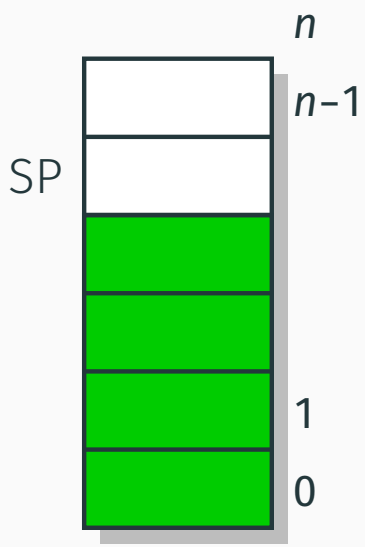
- pole délky n
- dno zásobníku na indexu 0
- stack pointer
  - index v poli, číslo typu `int`
  - neukazuje na poslední vložený prvek, nýbrž
  - ukazuje na **první volnou pozici** – dodržuje zvyklosti C++ o polích
  - prázdný zásobník – stack pointer roven 0
  - plný zásobník – stack pointer roven n

Pomocí spojových struktur – dynamická alokace paměti, pointery

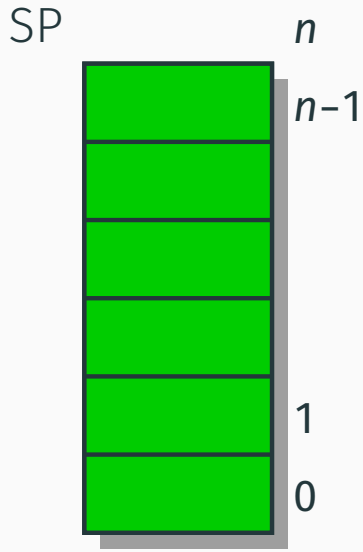
Zásobník – implementace pomocí pole



Prázdný



Částečně zaplněný



Plný

Zásobník – využití zásobníku

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

Zásobník – testování parity závorek

Zadání

Máme dán matematický výraz *`Expr`*, který obsahuje kulaté, hranaté a složené závorky. Úkolem je implementovat funkci, která bude vracet *`true`*, pokud je výraz správně uzávorkován, jinak bude vracet *`false`*.

Zásobník – testování parity závorek

Ukázka

Výraz *Expr*	Výsledek
`()`	`true`
`)(`	`false`
`(()())`	`true`
`[()()]`	`true`
`{()()()[()()]}`	`true`
`[()()`	`false`
`[()()]`	`false`
`[[()]`	`false`

Zásobník – testování parity závorek

Princip řešení

- Při průchodu řetězcem se musí pravé závorky objevovat v přesně opačném pořadí než levé závorky.
- Levé závorky je nutné si pamatovat tak, aby aktuální, posledně načtená, levá závorka byla zpracována jako první; zatímco první načtená levá závorka musí přijít na řadu až jako poslední.
- Zkontrolovaný pár závorek vyřadíme z dalšího zpracování.
- Způsob pamatování levých závorek přesně odpovídá mechanismu práce zásobníku – LIFO.

Zásobník – testování parity závorek

Input : Výraz *Expr* obsahující závorky ()[]{}
Output: *true* pokud je *Expr* správně uzávorkován, jinak *false*

```
1  Init (S);
2  foreach e_i ∈ Expr do
3      switch e_i do
4          case (, [, { do
5              Push (S, e_i);
6          end
7          case ), ], } do
8              if ¬IsEmpty(S) then
9                  if Pop (S) „není správná závorka“ then
10                     return false
11                 end
12             end
13             else
14                 return false
15             end
16         end
17     end
18 end
19 return IsEmpty (S)
```

Přílohy

Fronta

Fronta – možnosti implementace

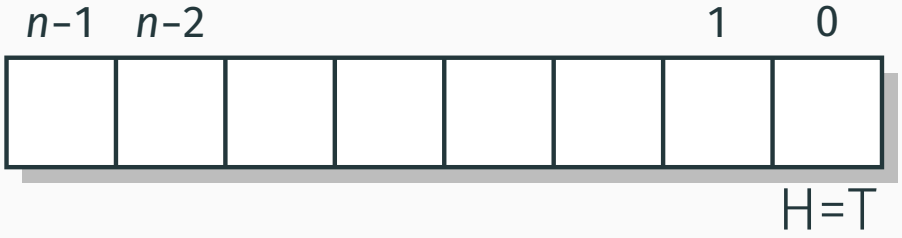
Pomocí spojových struktur – dynamická alokace paměti, pointery

Pomocí pole

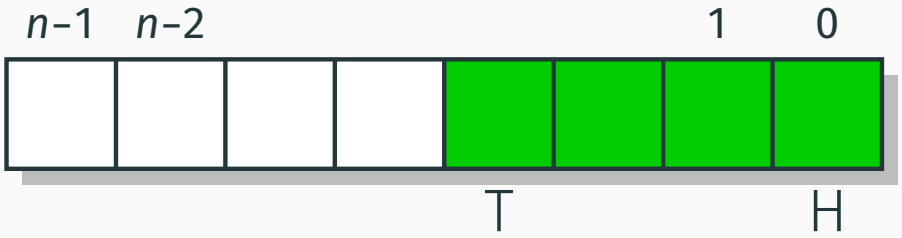
- hlava a ocas fronty jsou indexy do pole,
- jednoduché přidávání a odebírání – jak hlava, tak ocas se posunují dozadu.

Fronta – implementace v poli

Prázdná fronta

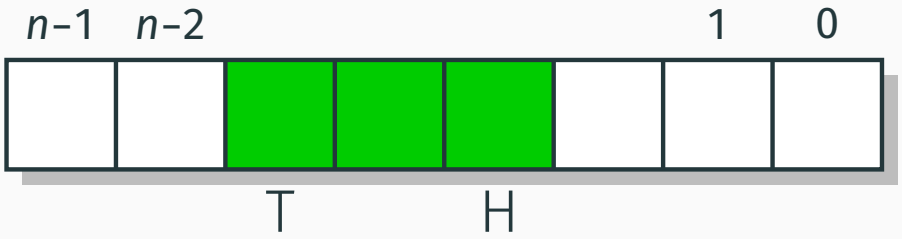


Vloženy 4 prvky

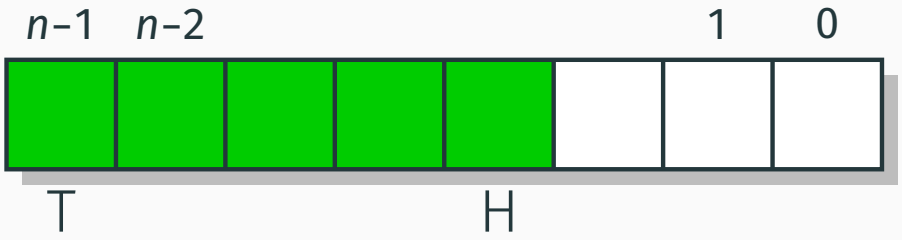


Vyjmuty 3 prvky, vloženy 2 prvky

Fronta – implementace v poli (pokrač.)



Vloženy 2 prvky



Výsledný stav:

- do fronty již nelze přidat další prvky,
- ale současně fronta není plná.

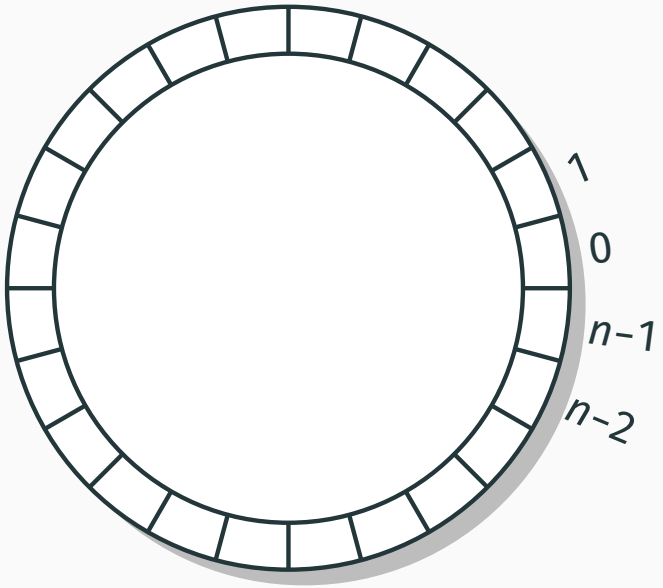
Fronta – implementace v poli (pokrač.)

Možná řešení:

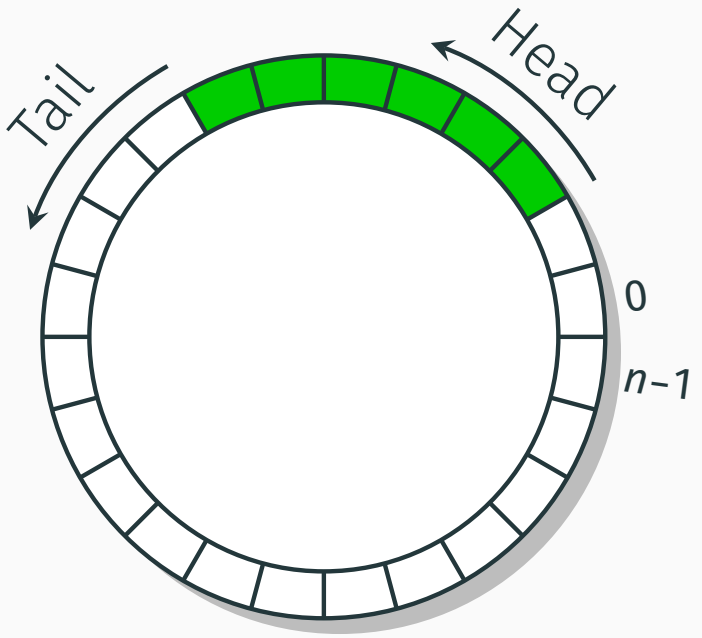
- přesouvat prvky na začátek pole – vyjmutí prvku již nebude operace s časovou složitostí $O(1)$ ale $O(n)$, bude docházet k masivnímu kopírování dat,

Fronta – kruhový buffer

Řešením je **kruhový buffer** (circular buffer) – začátek a konec pole se navzájem dotýkají.



Fronta – kruhový buffer



Charakteristika

- při vložení prvku se ocas fronty zvyšuje o jedničku,
- při odebírání prvku se hlava fronty zvyšuje o jedničku
- při překročení délky bufferu se hodnoty hlavy či ocasu vrací na první pozici tj. 0 – **modulární aritmetika**

Problém: Jak rozlišit prázdnou a zaplněnou frontu?

Řešení: V buffer je udržováno jedno prázdné místo. Ocas fronty ukazuje na první volnou pozici, ne na konec fronty.

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

Fronta – plánovač OS, cyklická obsluha (round robin)

Tato část OS přiděluje jednotlivým procesům sdílený procesor:

- vybere z fronty čekajících procesů první proces,
- obnoví registry procesoru,
- spustí obnovený proces,
- po uplynutí vymezeného časového kvanta, zahájí odebírání procesoru,
- uloží obsah registrů
- zařadí tento proces na konec fronty,
- vybere z fronty čekajících procesů první proces ...

Přílohy

Binární strom

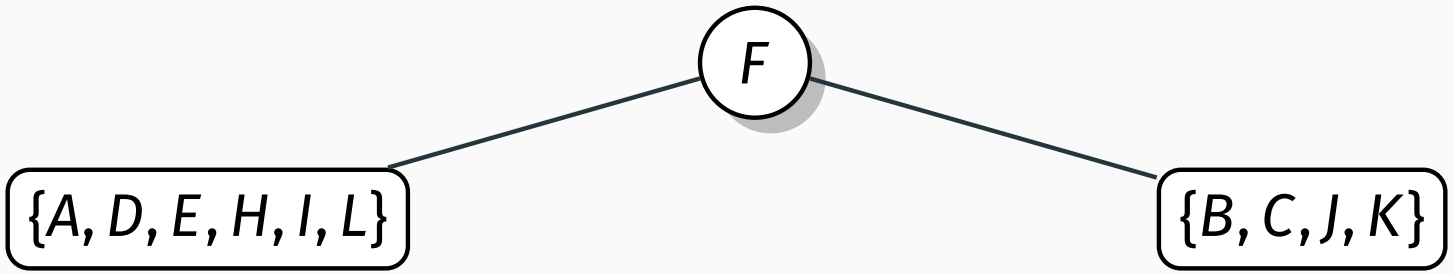
Budování binárního stromu podle rekurzivní definice

- Výchozí množina uzlů $M = \{A, B, C, D, E, F, H, I, J, K, L\}$
- Postupně budeme aplikovat pravidla z definice binárního stromu.
- Uzly ve tvaru kruhu odpovídají již vytvořeným uzlům binárního stromu, viz **Pravidlo 2**.
- Uzly ve formě množin odpovídají dosud nevytvořeným podstromům, viz **Pravidlo 2**. Zde budeme rekurzivně pokračovat v tvorbě binárního stromu.
- Prázdné množiny odpovídají prázdným podstromům, viz **Pravidlo 1**.

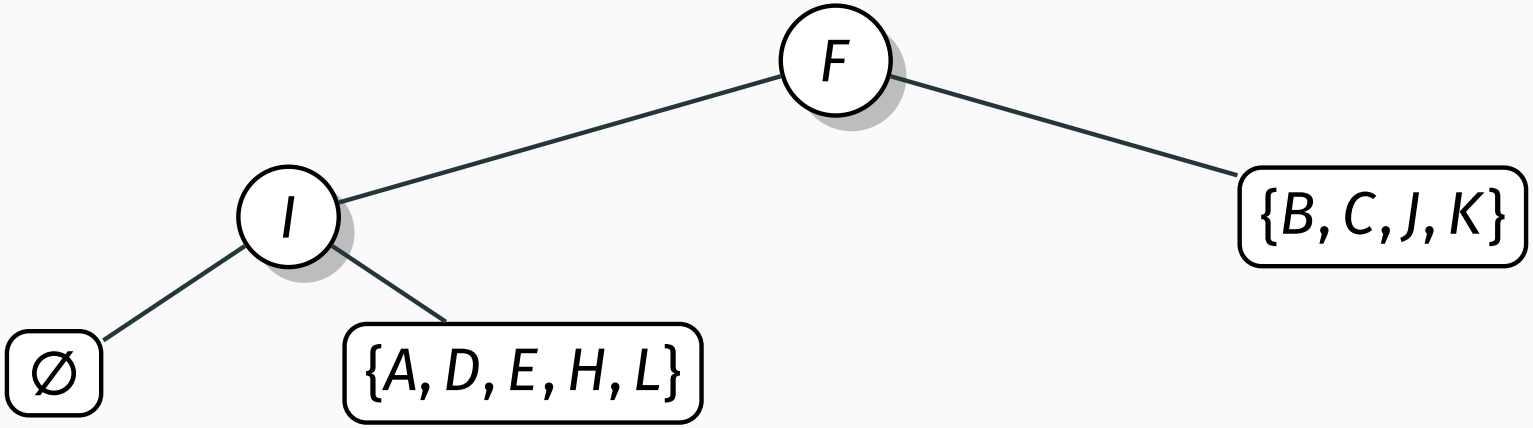
Budování binárního stromu podle rekurzivní definice

$\{A, B, C, D, E, F, H, I, J, K, L\}$

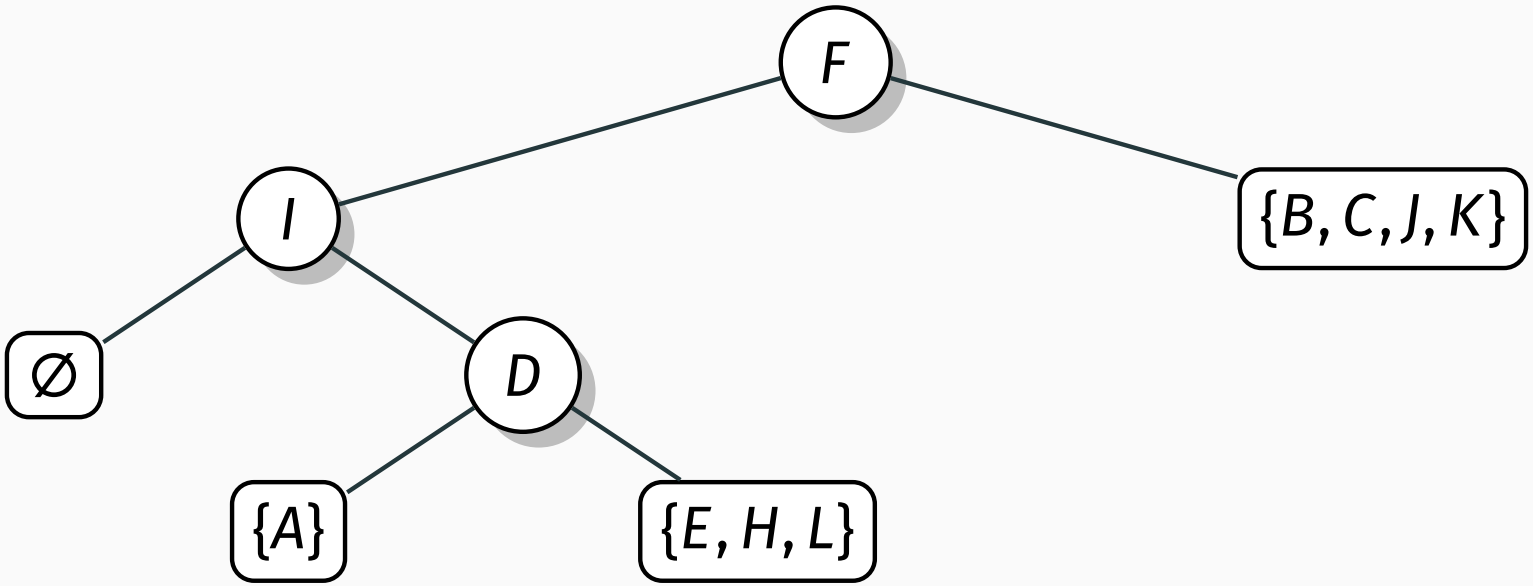
Budování binárního stromu podle rekurzivní definice



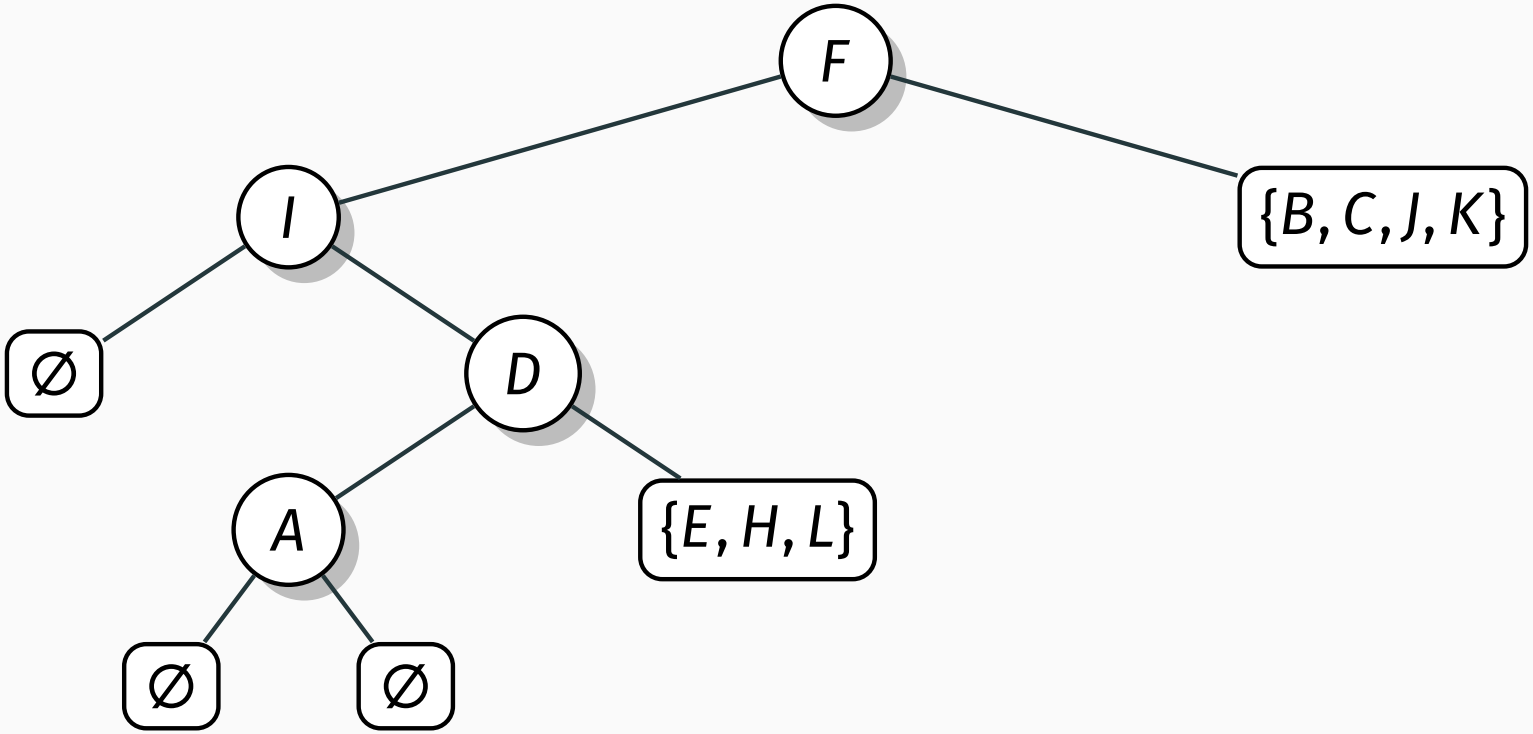
Budování binárního stromu podle rekurzivní definice



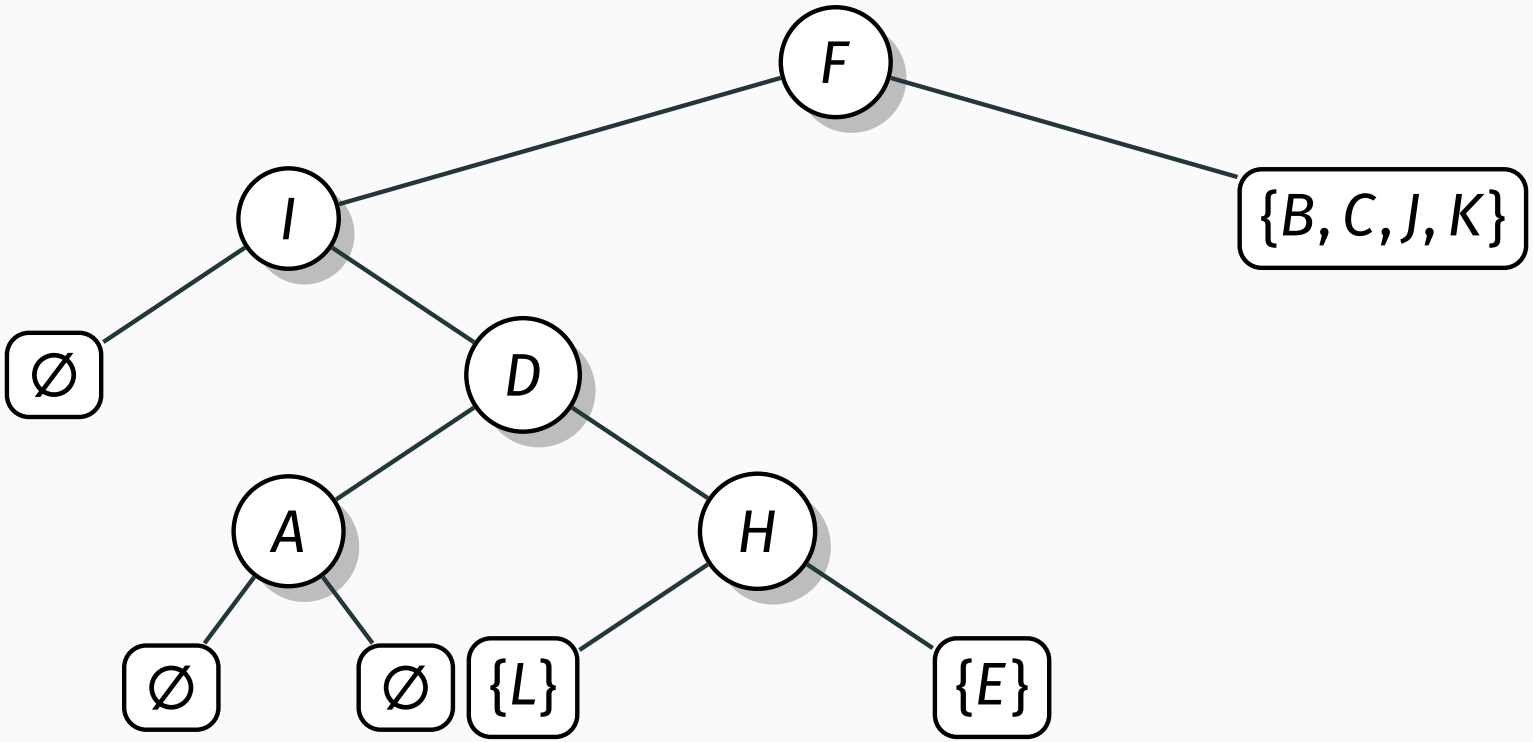
Budování binárního stromu podle rekurzivní definice



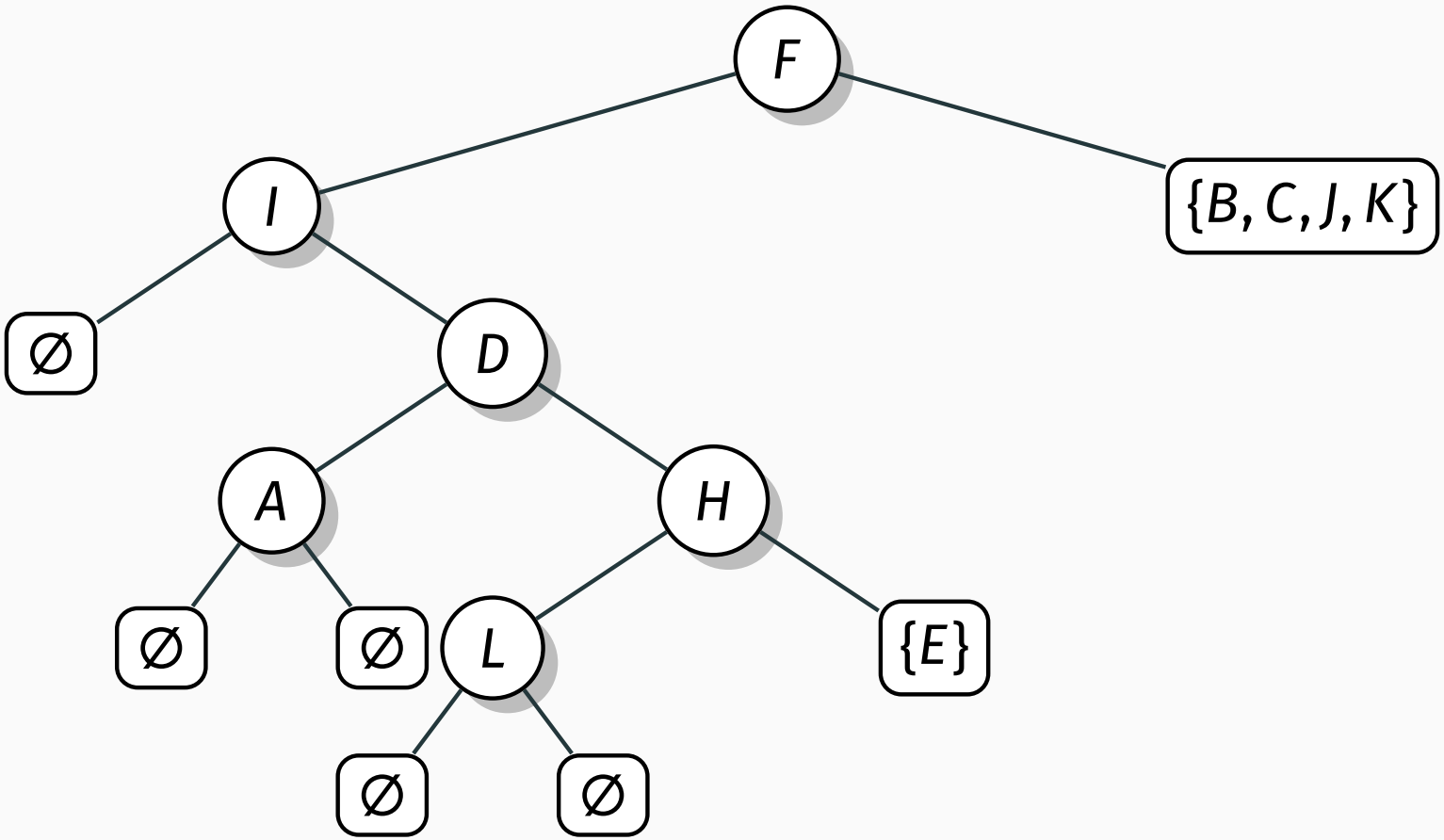
Budování binárního stromu podle rekurzivní definice



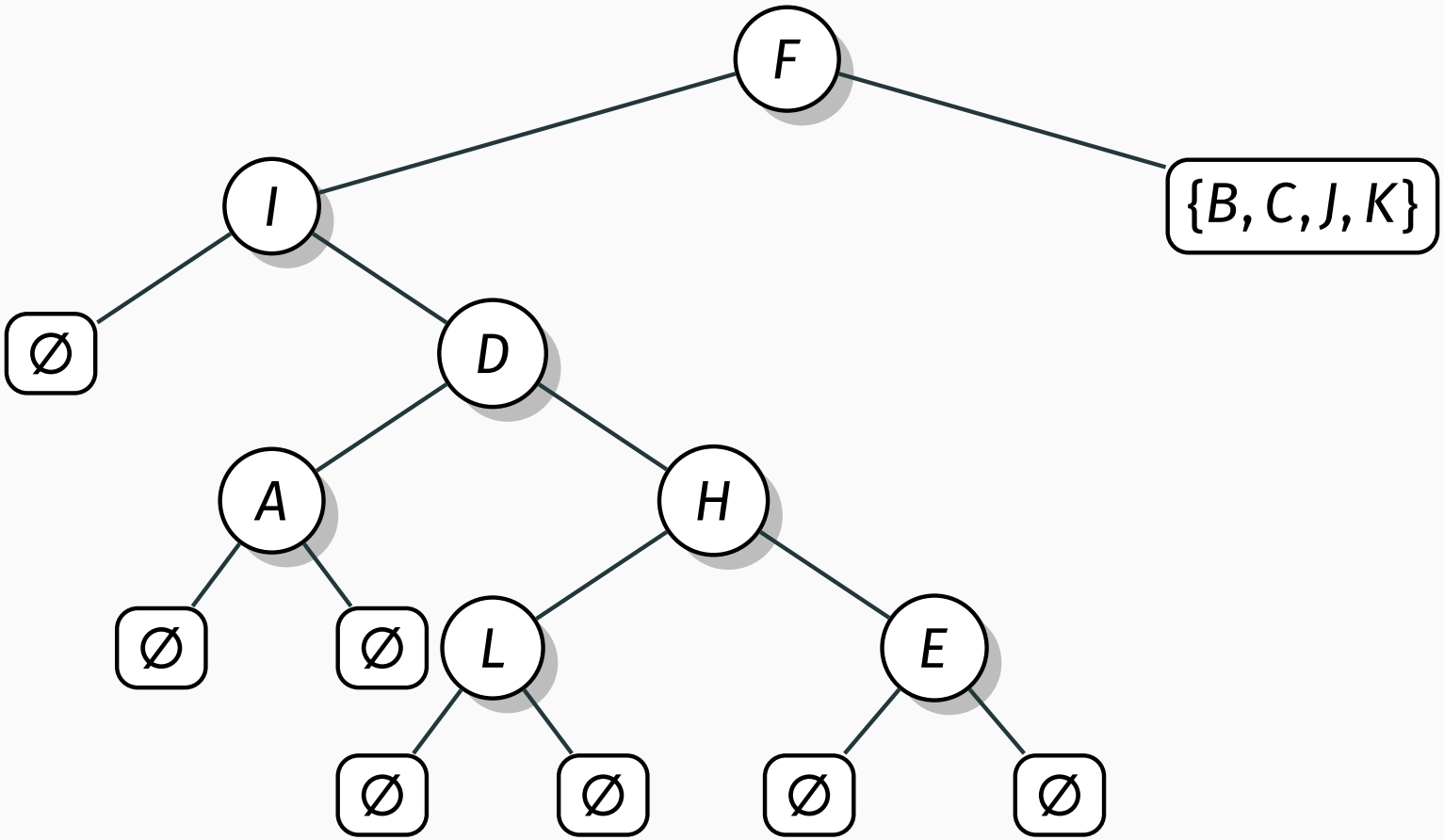
Budování binárního stromu podle rekurzivní definice



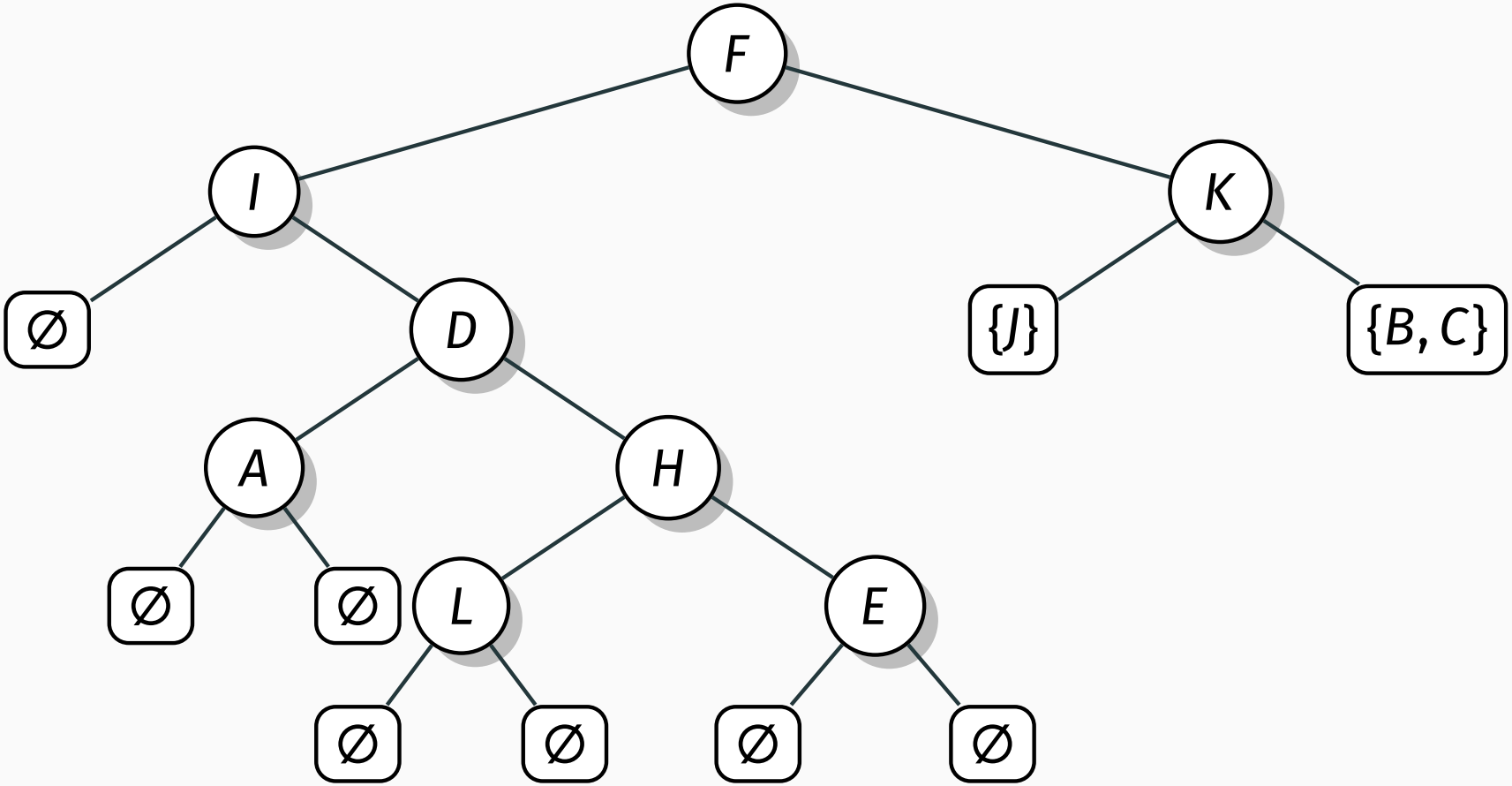
Budování binárního stromu podle rekurzivní definice



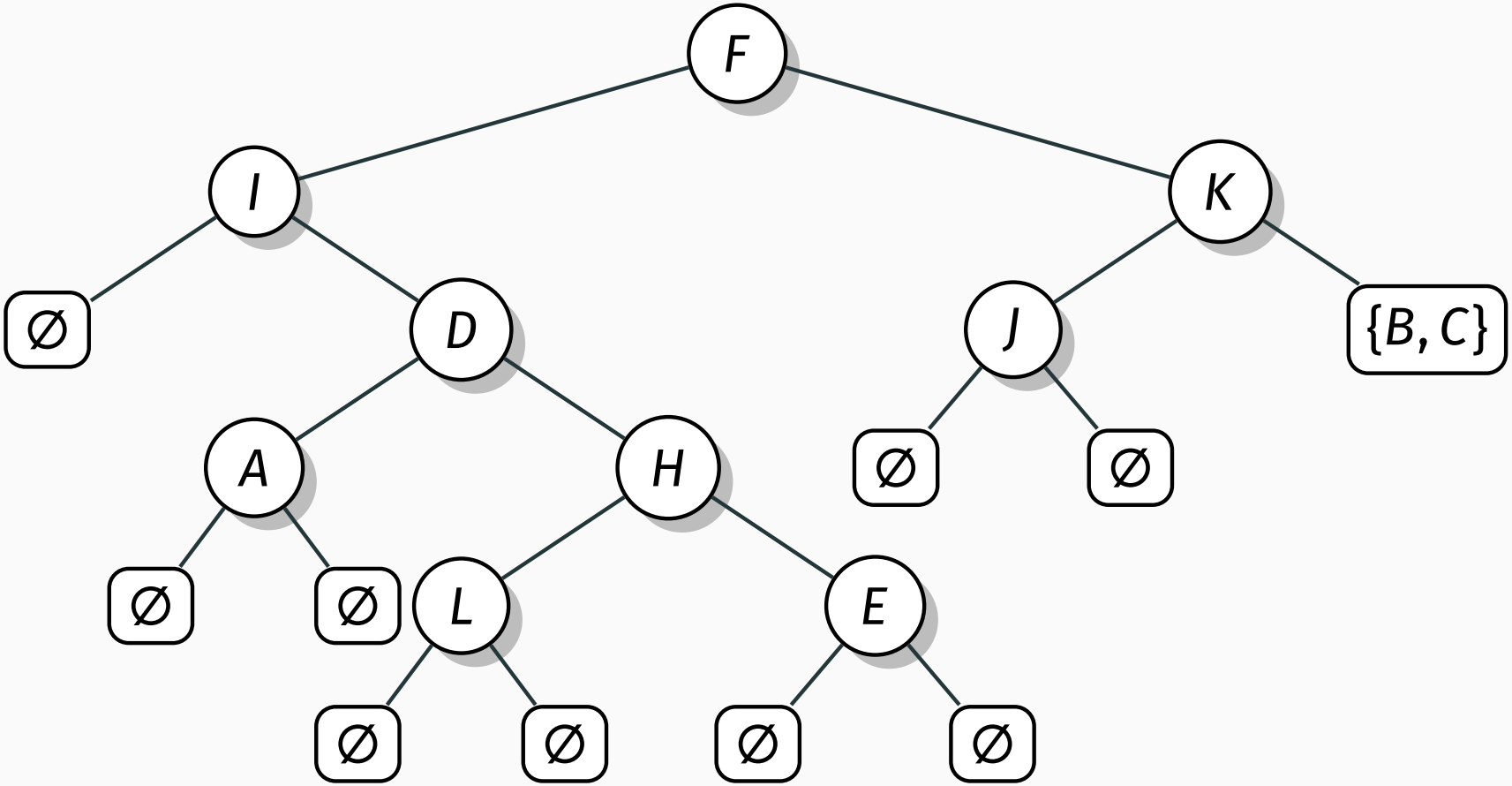
Budování binárního stromu podle rekurzivní definice



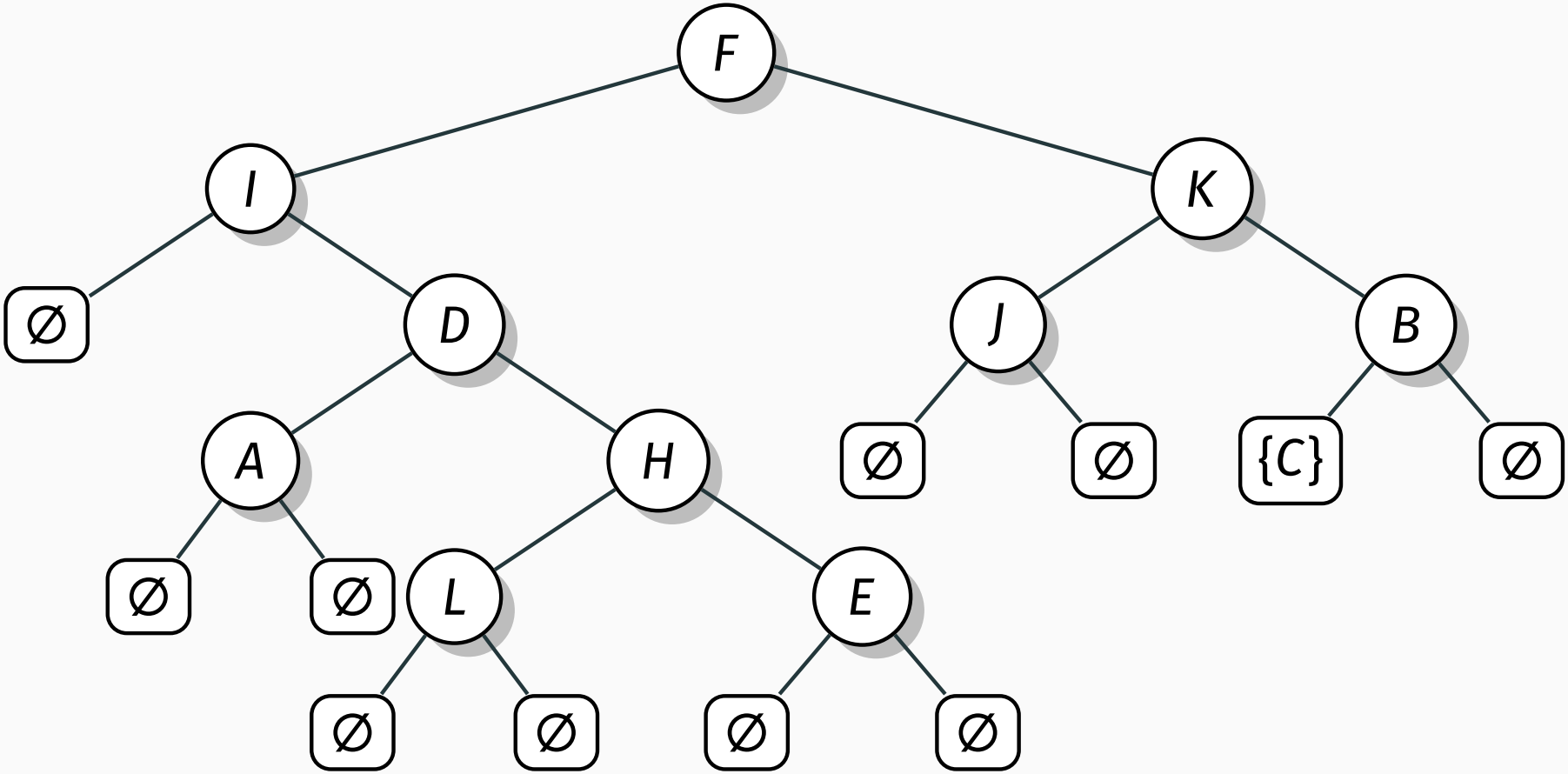
Budování binárního stromu podle rekurzivní definice



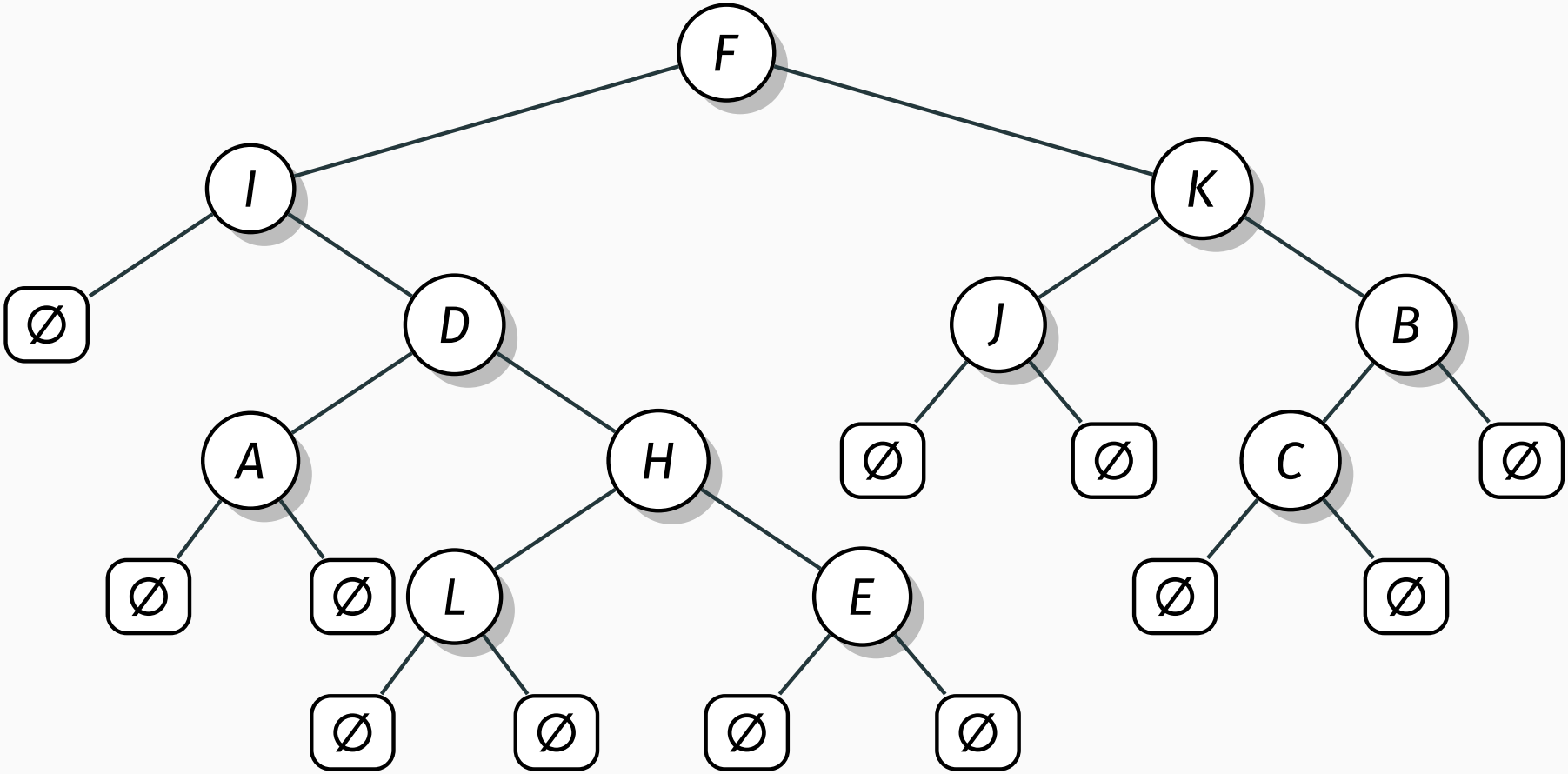
Budování binárního stromu podle rekurzivní definice



Budování binárního stromu podle rekurzivní definice

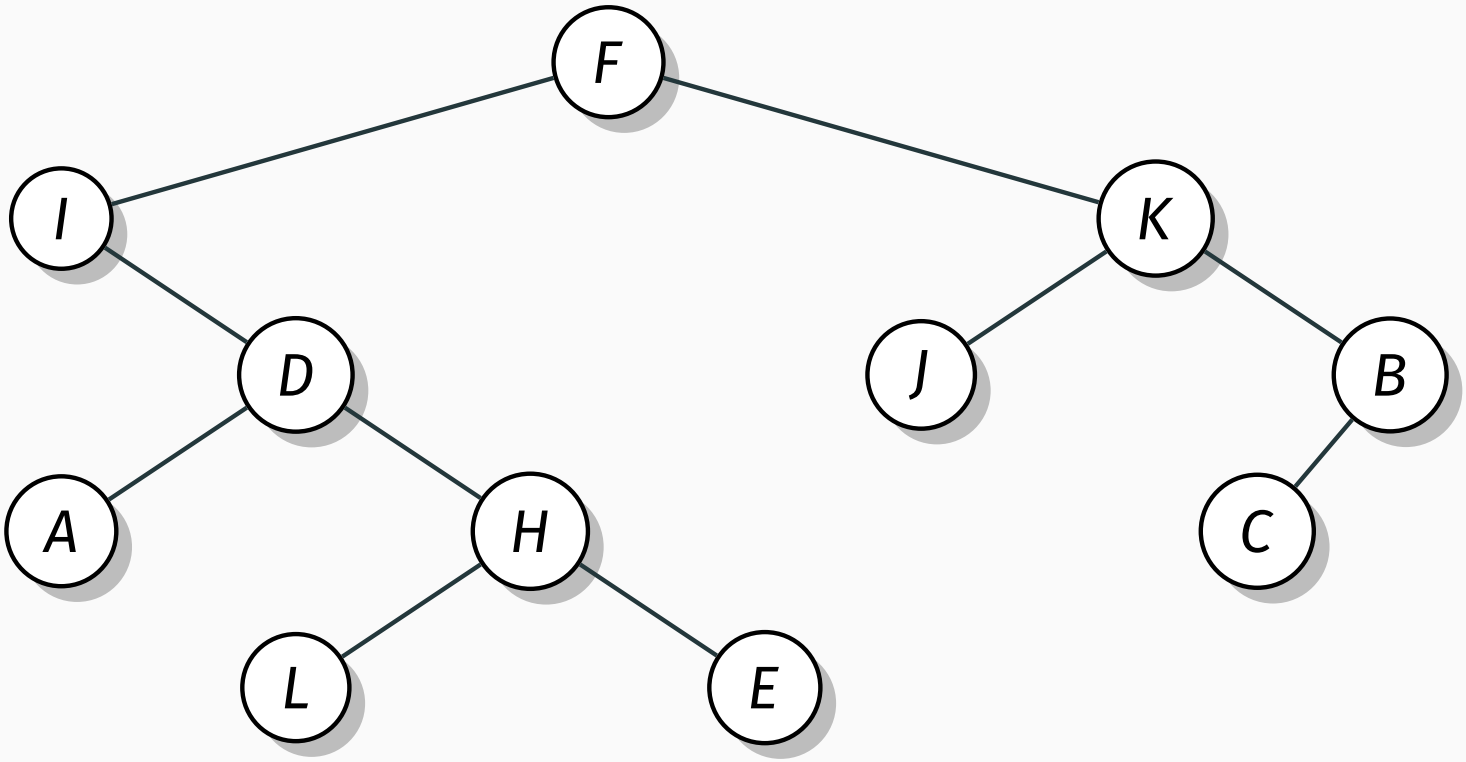


Budování binárního stromu podle rekurzivní definice



Budování binárního stromu podle rekurzivní definice

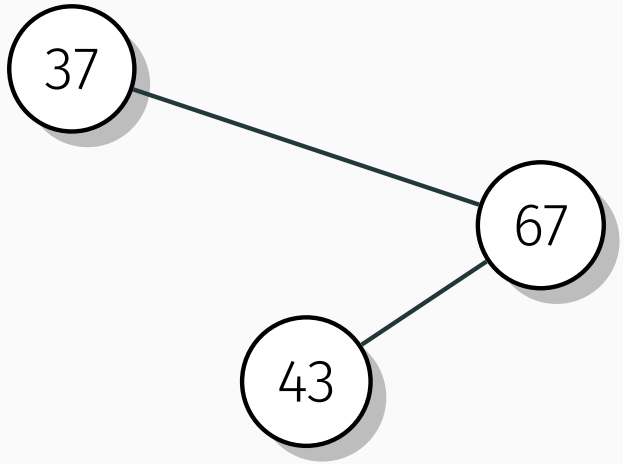
Odstraníme-li vizualizaci prázdných podstromů, zobrazených jako prázdné množiny, dostáváme výsledný binární strom v obvyklé podobě.



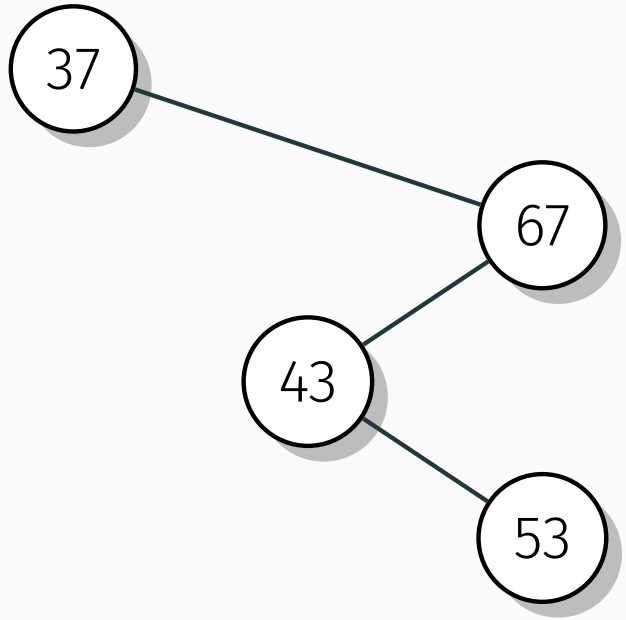
Binární vyhledávací strom – vložení klíče 37

Binární vyhledávací strom – vložení klíče 67

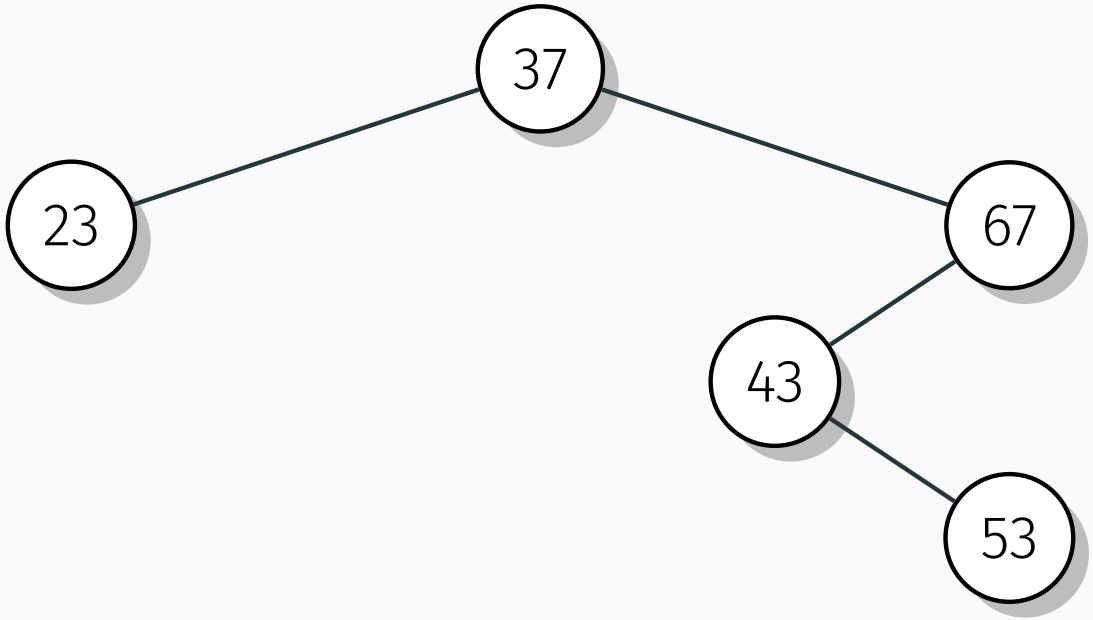
Binární vyhledávací strom – vložení klíče 43



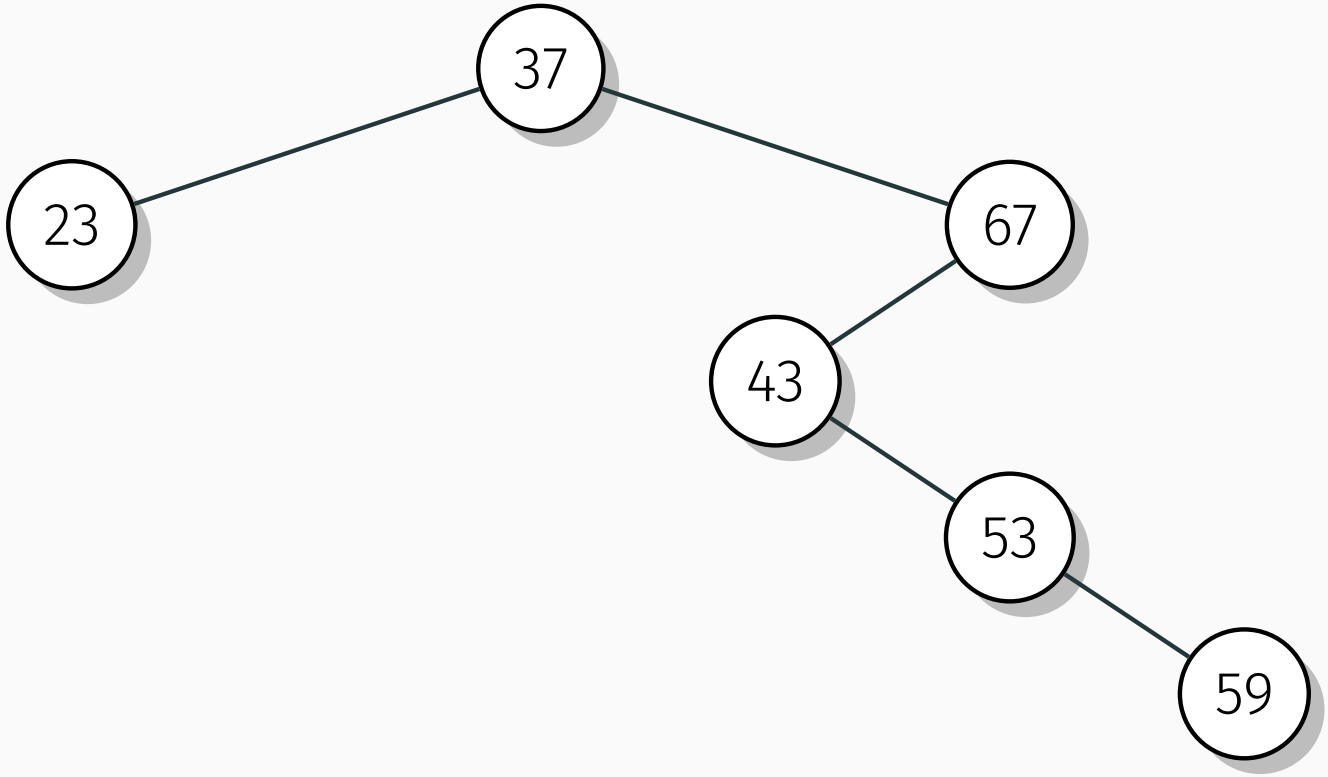
Binární vyhledávací strom – vložení klíče 53



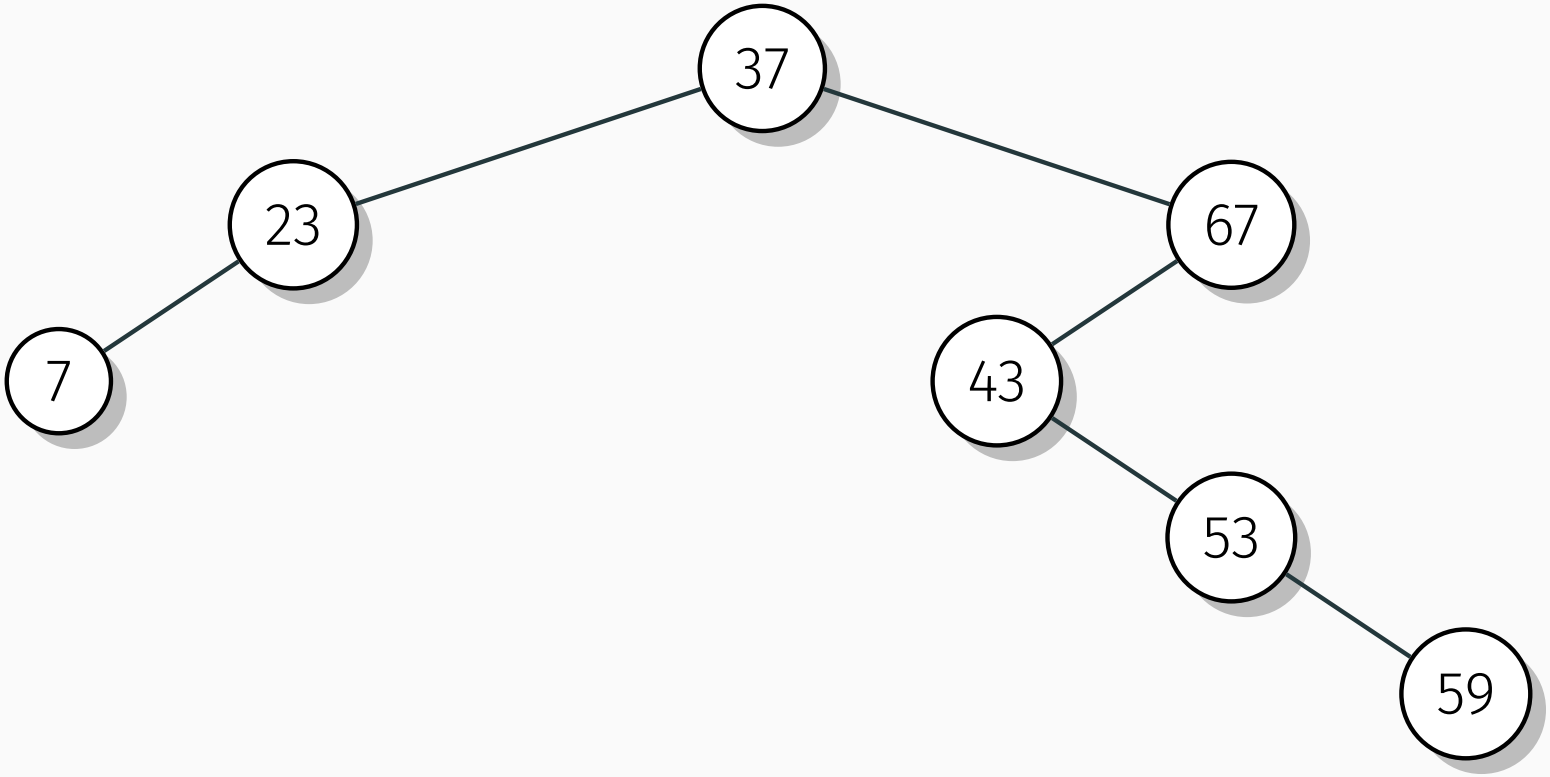
Binární vyhledávací strom – vložení klíče 23



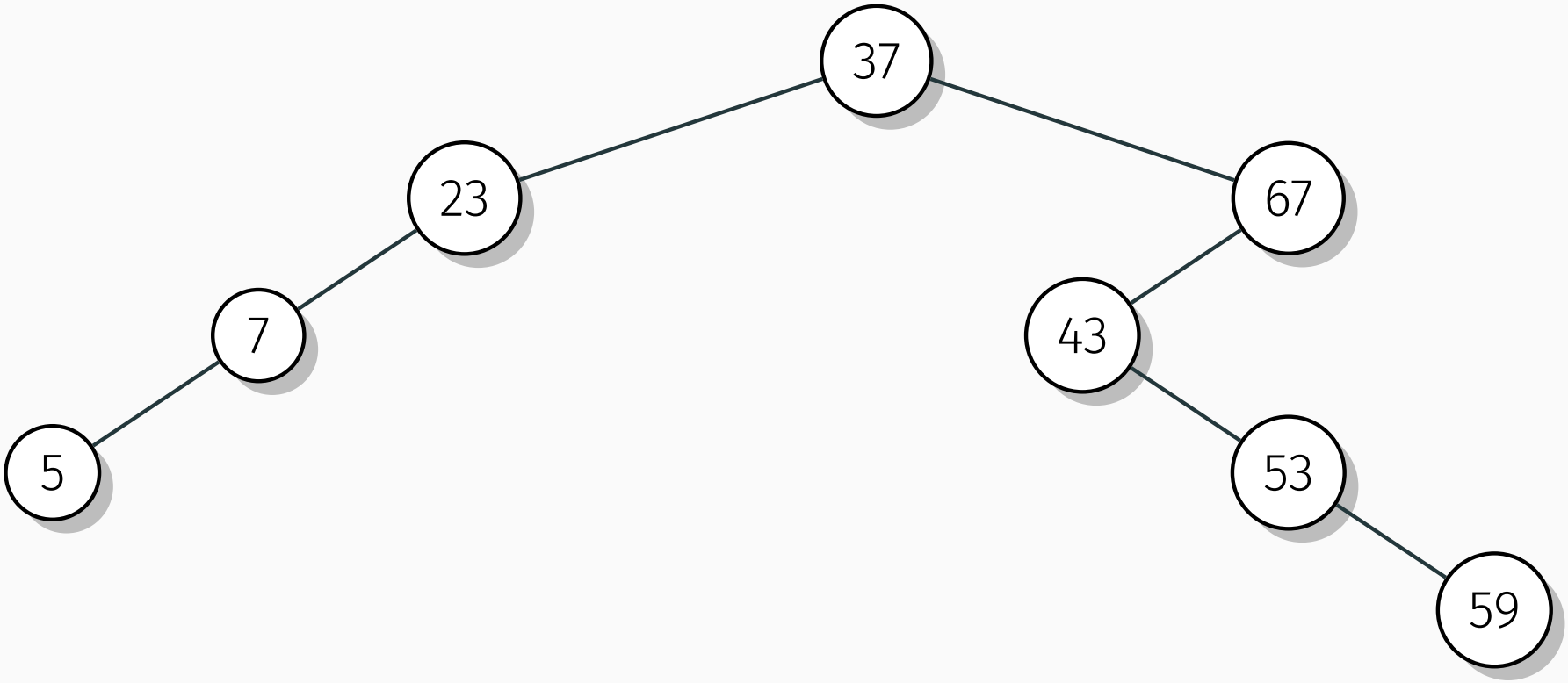
Binární vyhledávací strom – vložení klíče 59



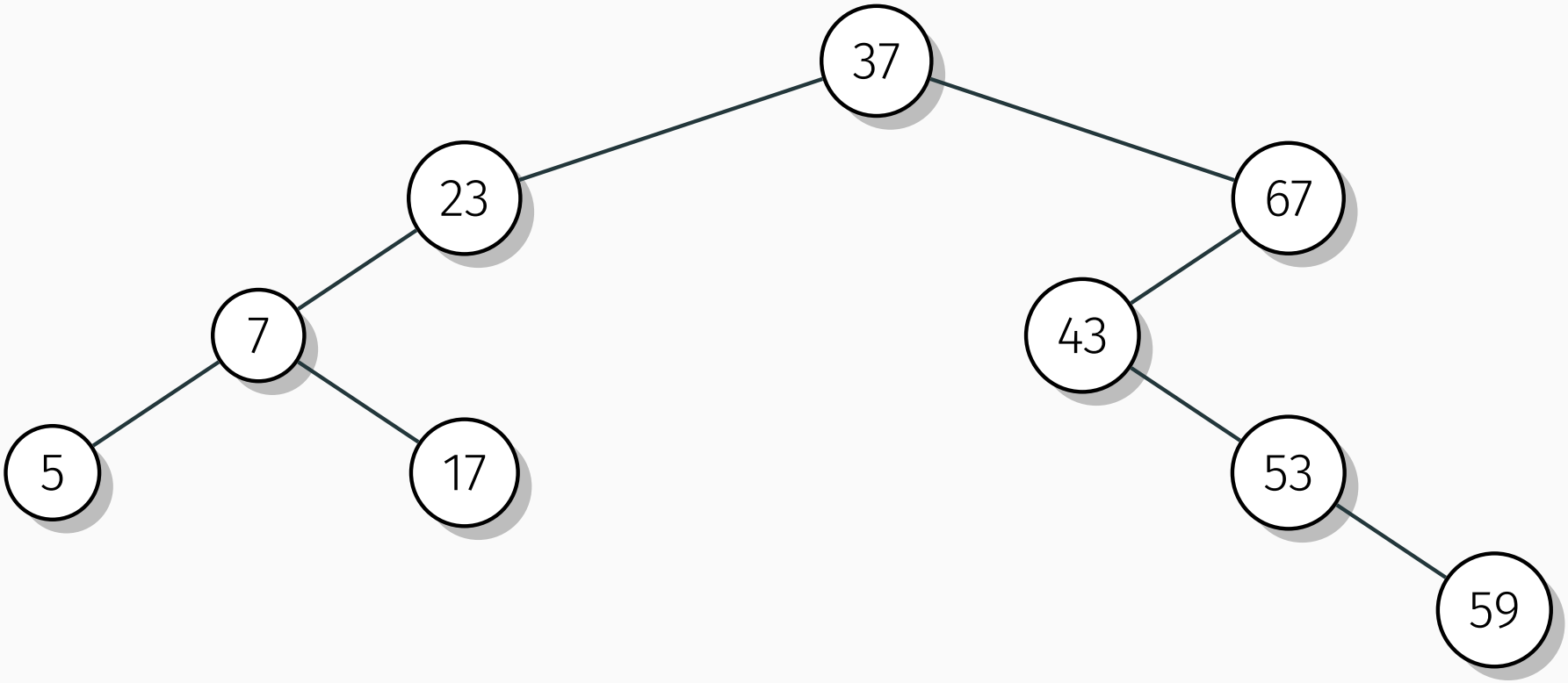
Binární vyhledávací strom – vložení klíče 7



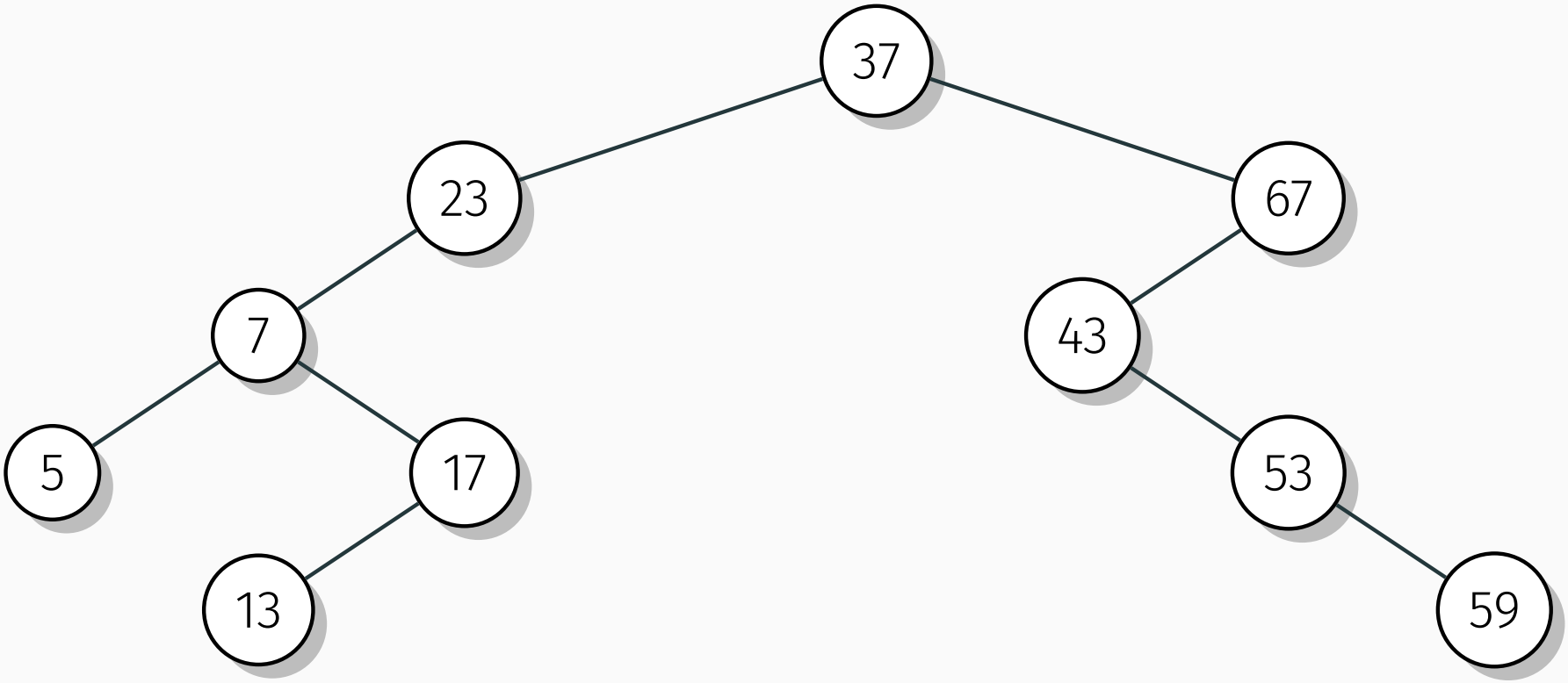
Binární vyhledávací strom – vložení klíče 5



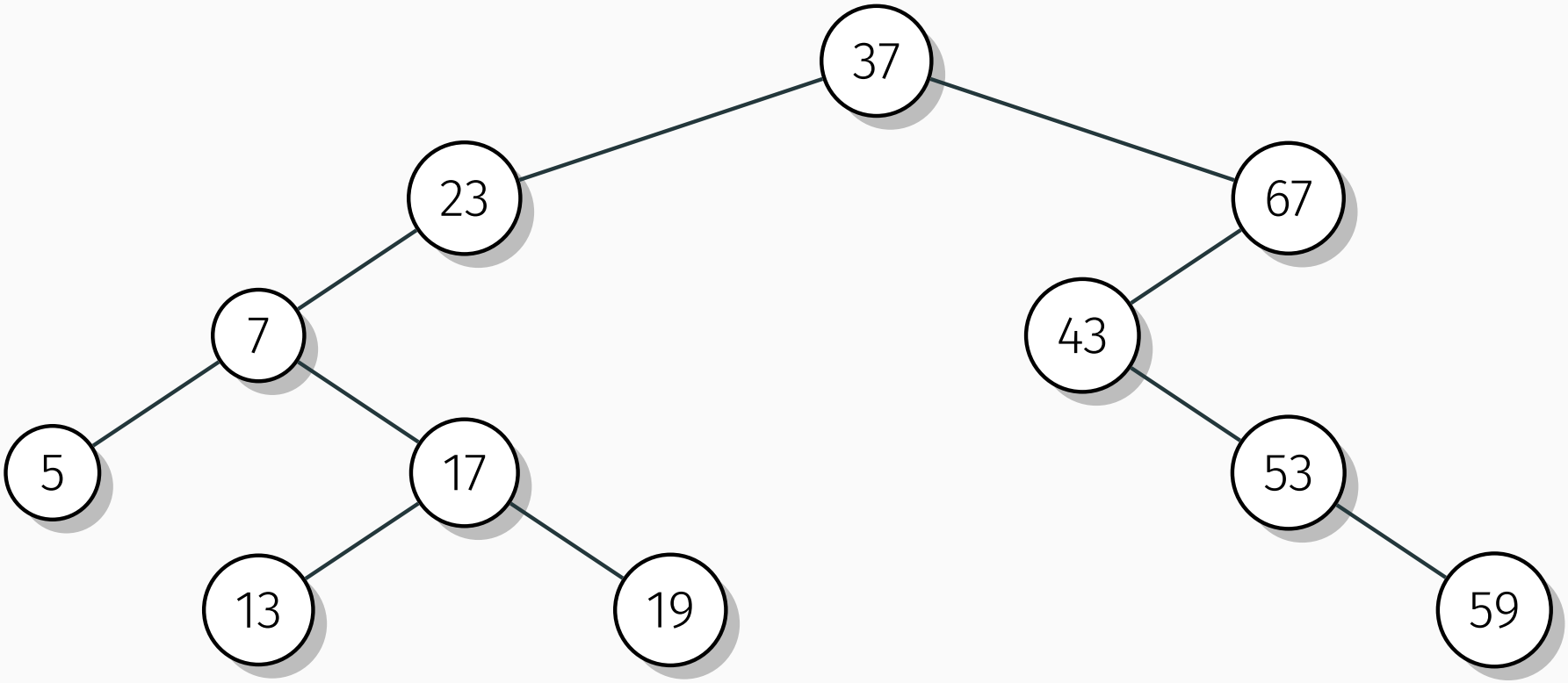
Binární vyhledávací strom – vložení klíče 17



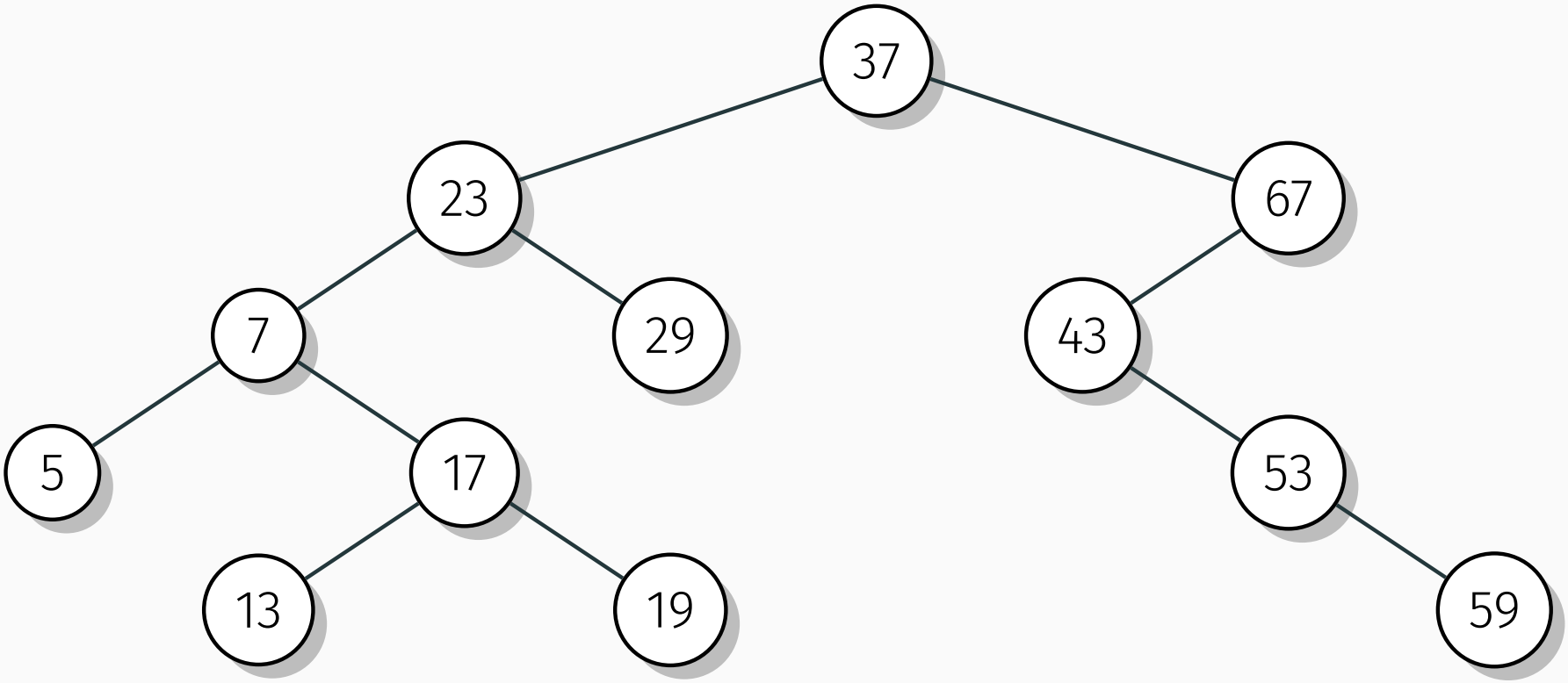
Binární vyhledávací strom – vložení klíče 13



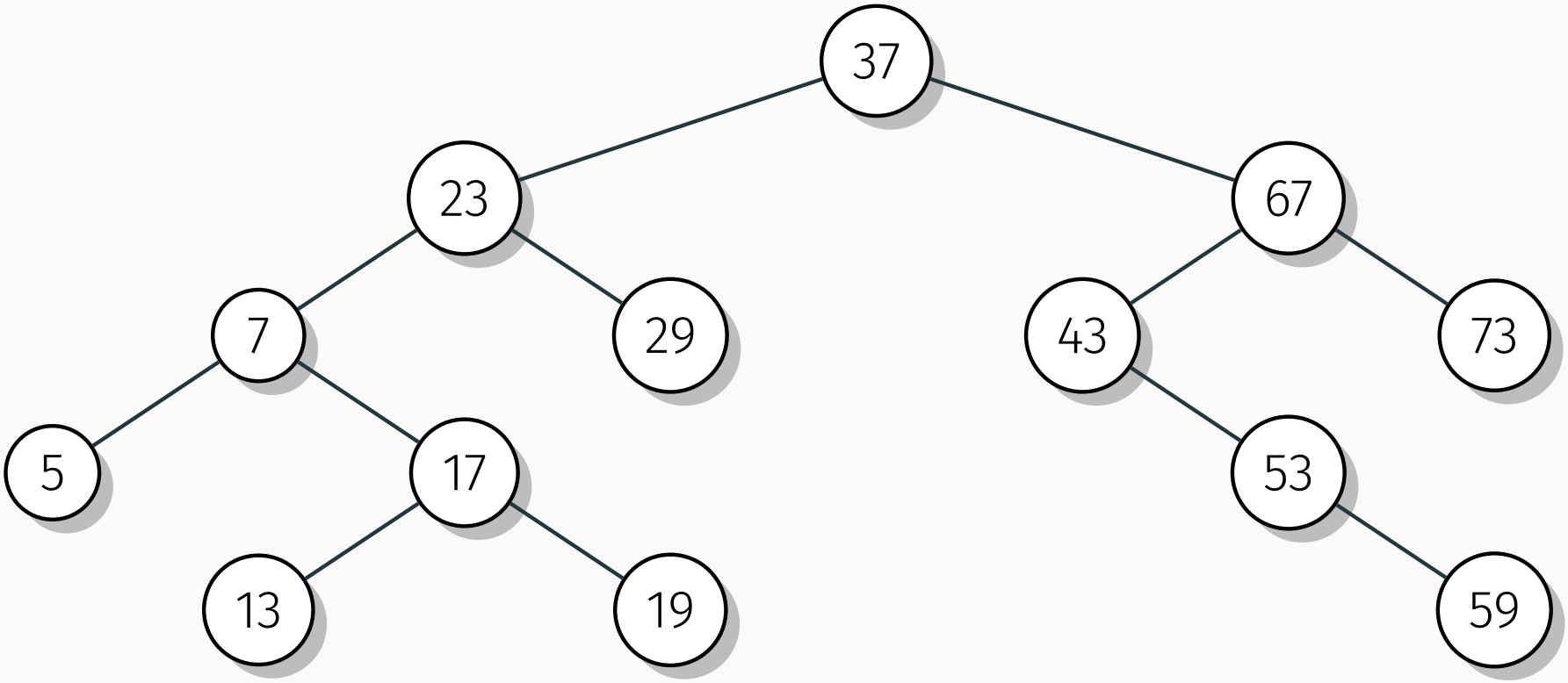
Binární vyhledávací strom – vložení klíče 19



Binární vyhledávací strom – vložení klíče 29



Binární vyhledávací strom – vložení klíče 73



Binární vyhledávací strom – vložení klíče 47