Algoritmy II – prezentace k přednáškám

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Prezentace ke dni 12. února 2020



Celková osnova všech přednášek

0. Úvodní přednáška
- O předmětu Algoritmy II
- Prezenční forma studia
  - Výuka
  - Úkoly a jejich hodnocení
- Kombinovaná forma studia
  - Výuka
  - Úkoly a jejich hodnocení
- Software pro výuku
- Studijní literatura

1. Lineární datové struktury
- Abstraktní datové struktury
- Zásobník
  - Popis
  - Implementace
  - Ukázka použití
- Fronta

Celková osnova všech přednášek (pokrač.)

  - Popis
  - Použití
  - Implementace

2. Dynamická alokace paměti
- Podobenství o žlutém lístečku a kastrolu s gulášem
- Podobenství o kabátu v šatně na plese
- Pointery v C++

3. Spojová implementace lineárních datových struktur
- Seznam
  - Popis
  - Použití
- Implementace obousměrného seznamu
  - Implementace pomocí pole
  - Spojová implementace

4. Grafy
- Historické úlohy z teorie grafů
- Neorientovaný graf

Celková osnova všech přednášek (pokrač.)

- Pojmy založené na cestách v grafu
- Způsoby zadání grafu
- Stromy

5. Algoritmy průchodů grafem
- Prohledávání grafu do šířky
- Prohledávání grafu do hloubky
- Aplikace
  - Hledání cesty v bludišti
  - Počítačová grafika – vyplňování oblasti

6. Binární vyhledávací stromy I
- Binární stromy
- Binární vyhledávací stromy

7. Binární vyhledávací stromy II

8. Vyvážené a vícecestné stromy
- B-stromy

9. Hašování

Celková osnova všech přednášek (pokrač.)

10. Vyhledávání v textu
    - Definice základních pojmů
    - Vyhledávání hrubou silou
    - Morris-Prattův algoritmus
    - Knuth-Morris-Prattův algoritmus

Úvodní přednáška

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



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

- Předmět je pokračováním předmětu Algoritmy I.
- Hlavní náplní předmětu jsou datové struktury a související algoritmy.
- V předmětu budou využívány znalosti z předmětu Programování II – objektově orientované programování.
- Probírané algoritmy a datové struktury budou demonstrovány v jazyce C++.
- Nemalý důraz je kladen na praktickou implementaci probíraných algoritmů a datových struktur. Studenti jsou vedeni k analýze algoritmizovaných problémů a k syntéze řešení z menších celků.

Prerekvizity

- **Prerekvizity** jsou soubor nutných podmínek, které musí student splňovat, aby si mohl daný předmět zapsat.
- Předmět Algoritmy II má jako povinnou prerekvizitu předmět Algoritmy I, bez úspěšného ukončení předmětu Algoritmy I není možné pokračovat v předmětu Algoritmy II.

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,
  - 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,

Garant 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. Pokud se nepodaří dosáhnout řešení problému s cvičícím obracejte se na garanta předmětu.

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.
- Rozhodně neplatí myšlenka: „Já k němu přijdu na konzultaci, on si o mě bude myslet, že jsem úplně blbej a u zápočtu mě vyhodí“.
- Přijdte se zeptat rovnou ke zdroji informací – internetová fóra jsou zaplevelena různými polopravdami i naprostými nesmysly.

Studenti se specifickými nároky

Centrum Slunečnice FEI

- `http://slunecnice-fei.vsb.cz/`,
- poskytování služeb a podpory 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 libovůli v termínech**.

Algoritmy II – výuka, úkoly a jejich hodnocení

Specifika výuky, úkolů 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.

Prezenční forma studia

Přednášky

- každé pondělí od 12:30 do 14:00, učebna NA1,
- slouží především k:
  - výkladu učiva,
  - zdůraznění klíčových poznatků či postupů v dané části učiva,
  - předvedení ukázkových příkladů, jejich typických řešení
  - a tak dále,
- na Vaše případné dotazy k přednášce jsem schopen odpovídat v konzultačních hodinách.

Stručná osnova přednášek

1. **Úvodní přednáška** – organizační záležitosti, souhrn nutných znalostí z předmětu Algoritmy I
2. **Lineární datové struktury** – abstraktní datové struktury, zásobník, fronta, seznam
3. **Dynamická alokace paměti** – pointery, operátory reference, dereference, alokování a dealokování paměti
4. **Spojová implementace lineárních datových struktur** – využití OOP a dynamicky alokovaných struktur
5. **Grafy** – základní pojmy, graf jako datová struktura, možnosti implementace grafu
6. **Algoritmy průchodů grafem** – průchod grafu do hloubky a do šířky, aplikace průchodu grafem
7. **Binární vyhledávací stromy I** – základní pojmy, vyhledávání

Stručná osnova přednášek (pokrač.)

8. **Binární vyhledávací stromy II** – vkládání, rušení vrcholů, průchody stromem
9. **Vyvážené a vícecestné stromy** – AVL-stromy, Red-black stromy. B-stromy
10. **Hašování** – hašovací tabulky, metody řešení kolizí
11. **Vyhledávání v textu** – vyhledávání jednoho a více vzorků, elementární lexikální analýza textu
12. **Jednoduchý překladač** – překlad aritmetických a logických výrazů, postfixová notace a její interpretace pomocí zásobníku
13. **Techniky řešení problémů** – rozděl a panuj, žravý algoritmus, dynamické programování

Cvičení

- Náplní cvičení je implementace několika příkladů vztahujících se k probíranému učivu.
- Dále jsou na cvičeních obhajovány semestrální projekty.
- Vaše aktivita na cvičeních je průběžně hodnocena.
- 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. Nelze psát testy na jiných cvičeních, než která máte zapsaná v Edisonu.

Cvičení

Cvičení nenahrazuje přednášku!

- Na cvičeních je možné se cvičícím konzultovat učivo, ale
- neočekávejte, že pokud nebudete chodit na přednášky, tak Vám cvičící na cvičeních bude dělat jakousi bleskovou náhradní přednášku, abyste se vůbec mohli pustit do příkladů, jejichž řešení se na cvičení předpokládá.
- Bývá dobrým zvykem, že studenti se na cvičení aspoň minimálně připraví. Není nutné látku precizně ovládat, ale je nutné se orientovat v základních pojmech. Jinak cvičení nemají smysl.

Úkoly a jejich hodnocení

- 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.
- Za každý úkoly 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.
- 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.

Úkoly – průběžná aktivita na cvičeních (pokrač.)

- Z takto získaných vah z jednotlivých cvičení se na konci semestru vypočte průměrná váha, která se vynásobí maximálním možným počtem bodů (20) a výsledek je Vámi získaný počet bodů.
- Je zřejmé, že všechny zelené kódy odpovídají maximálnímu počtu bodů (20), samé červené kódy odpovídají nulovému počtu bodů.
- Body za aktivitu nelze získat zpětně.

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 $0,65 \times 20 = 13$ bodů.

Úkoly – projekt

- V průběhu semestru každý student vypracuje projekt. Zadání témat, vypracování a obhajoby proběhnou ve třech „dávkách“. Každé dávky se zúčastní třetina studentů.
- Na webu předmětu bude zveřejněno rozdělení studentů do dávek a odpovídající zadání.
- Tři cvičení během semestru budou věnována obhajobám projektů.
- Na každém z těchto tří cvičení vystoupí určená třetina studentů se svým projektem a seznámí cvičícího a své spolužáky s řešením zadaného projektu.
- Odevzdání projektu proběhne prostým přinesením projektu na cvičení, není nutné projekt zasílat dopředu.
- Pokud vznikne pochybnost o autorství projektu, obhajoba bude hodnocena 0 body. Oprava bude realizována formou testu z programování ve zkouškovém období.

Úkoly – písemná práce

- Řádný termín závěrečné písemná práce proběhne **přednášce 13. května 2019**.
- Opravné termíny proběhnou ve zkouškovém období.
- Řádný termín závěrečné písemné práce je pro všechny studenty **povinný**.
- Pokud se řádného termínu nezúčastníte, je vaší povinností se garantovi předmětu **předem omluvit**. Termín „předem“ znamená nejpozději do začátku daného termínu.
- Pokud se neomluvíte, daný úkol bude považován za nesplněný (0 bodů) a opravný termín vám nebude umožněn. Důsledkem je pak neúspěšné ukončení celého předmětu.
- Opravný termín na závěrečnou písemnou práci je poskytován jen těm studentům, kteří u svého prvního pokusu získali aspoň 10 bodů.

Úkoly – písemná práce (pokrač.)

Počet bodů na prvním termínu	Opravný termín
0 až 9	**NE**
10 až 19	ANO
více než 20	není nutný, úspěch

- Obdobná povinnost omluvit se platí i pro opravný termín.
- Na tento úkol máte **jeden řádný pokus** a **jeden opravný pokus**. Žádné další pokusy nejsou možné.

Hodnocení úkolů

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

Úkol	Min. počet bodů	Max. počet bodů
Průběžný test znalostí	10	20
Obhajoba projektu	21	40
Písemná práce	20	40
Celkem	**51**	**100**

Konec části pro prezenční formu studia

Kombinovaná forma studia

Tutoriály

1. tutoriál

- 15. února 2019
- Na tomto úvodním soustředění Vám budou sděleny informace o organizaci studia předmětu a informace o náplni předmětu.

2. tutoriál

- 1. března 2019
- K tomuto datu se předpokládá zvládnutí následujících témat: Abstraktní datové struktury. Zásobník, fronta, seznam. Abstraktní datové struktury s využitím OOP.

3. tutoriál

- 15. března 2019
- K tomuto datu se předpokládá zvládnutí následujících témat: Grafy. Graf jako datová struktura. Možnosti implementace grafu. Průchod grafu do hloubky a do šířky. Aplikace průchodu grafem.

Tutoriály (pokrač.)

4. tutoriál

- 29. března 2019
- povinný tutoriál
- K tomuto datu se předpokládá zvládnutí následujících témat: Binární stromy, definice, vyhledávání, vkládání, rušení vrcholů, průchody stromem. Na tomto tutoriálu proběhne první test.

5. tutoriál

- 13. dubna 2019
- povinný tutoriál
- Tento tutoriál je rozdělen do několika skupin, tutoriál proběhne na počítačové učebně, celou náplň tutoriálu bude tvořit druhý test.

Tutoriály (pokrač.)

6. tutoriál

- 3. května 2019
- Není

7. tutoriál

- 17. května 2019
- povinný tutoriál
- K tomuto datu se předpokládá zvládnutí následujících témat: Vyvážené binární stromy. B-stromy. Hašování. Konzultace k projektu.

Úkoly – první test

- První test se bude psát na tutoriálu **29. března 2019**.
- Bude to krátký písemný test na cca 10 až 15 minut maximálně.

Úkoly – druhý test

- Druhý test se bude psát na tutoriálu **13. dubna 2019**.
- Opravný termín bude ve zkouškovém období.
- Na začátku vymezené doby dostanete zadání, které ve zbývajícím čase naprogramujete, odladíte a předvedete.

Úkoly – třetí test

- Třetí test uvedený v Edisonu je zrušen.
- Body přesunuty do projektu.

Úkoly – projekt

- Koncem března budou na webu tutora zveřejněna zadání projektů a jejich rozdělení mezi studenty.
- Deadline pro odevzdání projektu je **19. května 2019 23:59**.
- Podrobné pokyny k odevzdání budou zveřejněny se zadáním.
- Ve zkouškovém období proběhne obhajoba.
- Termíny budou vypsány v systému Edison.
- K obhajobě projektu není opravný termín.

Hodnocení úkolů

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

Úkol	Min. počet bodů	Max. počet bodů
První test	10	20
Druhý test	10	20
Projekt	31	60
Celkem	**51**	**100**

Obecné poznámky k 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, přenášet elektronickým způsobem a podobně.
- Předmět je ukončen klasifikovaným zápočtem. Nevztahuje se tudíž na něj požadavek dvou opravných pokusů, jak to vyžaduje studijní řád u zkoušky.

Pokyny k prvnímu a druhému testu

- Na každý z těchto úkolů máte **jeden řádný pokus** a **jeden opravný pokus**.
- Na všechny tři úkoly budou v Edisonu vypsány řádné termíny, kde se budete hlásit. Na první test bude vypsán jeden řádný termín. Na druhý test bude v jeden den vypsáno několik řádných termínů (kapacita počítačových učeben je menší, test musí běžet na několik „kol“).
- Absolvování prvního, druhého a třetího testu v řádném termín tj. na příslušném tutoriálu je **povinné**.
- Pokud se řádného termínu testu nezúčastníte, bez ohledu na to zda jste byli přihlášeni nebo ne, je vaší povinností se garantovi předmětu **předem** omluvit. Termín „předem“ znamená nejpozději do začátku termínu, na kterém máte daný test psát. Postačující je omluva emailem na adresu garanta předmětu.

Pokyny k prvnímu a druhému testu (pokrač.)

- Pokud se neomluvíte, daný úkol bude považován za nesplněný (0 bodů) a opravný termín vám nebude umožněn. Důsledkem je pak neúspěšné ukončení celého předmětu.
- Pokud se řádného termínu zúčastníte a neuspějete s řešením zadaného úkolu, máte nárok na opravný termín.
- Obdobná povinnost omluvit se platí i pro opravný termín.
- Opravné termíny budou vypsány ve zkouškovém období.

Konec části pro kombinovanou formu studia

Software pro výuku

Primární software:

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

Doplňkový software:

- Dokumentační systém Doxygen, `www.doxygen.org`
- Typografický systém LaTeX, `www.cstug.cz`

Vývojová prostředí pro C++

- Na učebnách je pro výuku k dispozici Microsoft Visual Studio 2017 (VS2017).
- Pro domácí přípravu si stáhněte Visual Studio Community 2017 (VS2017) a to buď prostřednictvím:
  - Microsoft Imagine, `http://elms.cs.vsb.cz/` nebo
  - přímo ze stránek Visual Studio Community, `https://www.visualstudio.com/cs/vs/community/`.
- Vývojové prostředí VS2017 spolu s integrovaným překladačem jazyka C++ bude užíváno jednak pro **výuku** a jednak jako **referenční překladač** při hodnocení Vašich případných projektů.

Studijní literatura

Učební materiály lze rozdělit do dvou skupin:

- literatura o algoritmech a datových strukturách a
- literatura o programovacím jazyku C++.

Níže uvedenou literaturu využijete v předmětech:

- Algoritmy I a Algoritmy II,
- Programování I a Programování II.

Základní literatura o algoritmech

1. Dvorský J.: **Algoritmy**, učební text, studijní opora v terminologii kombinované formy studia, dostupné na `http://www.cs.vsb.cz/dvorsky/Opora.html`
2. Sedgewick R.: **Algoritmy v C**, části 1-4, SoftPress, Praha, 2003. *Existuje i v anglické verzi, náročná, ale vynikající kniha.*
3. Wróblewski P.: **Algoritmy. Datové struktury a programovací techniky**, Computer Press, Praha 2003, druhé vydání 2015
4. Cormen, Leiserson, Rievest: **Introduction to Algorithms**, MIT Press, 2001.
5. Wirth, N.: **Algoritmy a štruktúry údajov**, Alfa, Bratislava, 1989

Doplňková literatura o algoritmech

1. Topfer, P.: Algoritmy a programovací techniky, Prometheus, Praha 1995.
2. Virius, M.: Základy algoritmizace, ČVUT Praha, 1997, skripta.
3. Honzík, J. a kolektiv: Programovací techniky, VUT Brno, 1987, skripta.
4. Harel, D.: Algorithmics, The Spirits of Computing, Addison-Wesley Publishing Company, 1993.
5. Sedgewick, R.: Algorithms in C++, Addison-Wesley Publishing Company, 1992.
6. Wood, D.: Data Structures, Algorithms and Performance, Addison-Wesley Publishing Company, 1993.

Základní literatura o C++

O jazyce C++ existuje v dnešní době řada česky psaných knih. Následující knihy můžete využít pro své studium:

1. Liberty J., Jones B. L.: **Naučte se C++ za 21 dní**, 2. aktualizované vydání, ComputerPress, 2007, ISBN 978-80-251-1583-1. *Zdařilá učebnice jazyka C++. Kniha je sice poněkud tlustá, nicméně je rozdělena do 21 lekcí rozumného rozsahu. Pokrývá veškerou látku, kterou bychom měli za oba semestry probrat.*
2. Eckel B.: **Myslíme v jazyku C++**, Grada Publishing, 2000, ISBN 80-247-9009-2. *Kniha od klasika ve výuce jazyka C++. Opět pokrývá veškerou látku, kterou bychom měli za oba semestry probrat.*. Kniha je dostupná v angličtině i online na serveru `http://www.mindviewinc.com/Index.php`
3. Stroustrup, B.: **C++ Programovací jazyk**. Česky: BEN-technická literatura, Praha 1997. *Vynikající kniha přímo od autora jazyka C++. Kniha dosti rozsáhlá, ale na druhou stranu obsahující naprosto vše co lze o C++ napsat.*

Doplňková literatura o C++

Dále je možné čerpat zajímavé informace z následujících knih:

1. Virius M.: **Pasti a propasti jazyka C++**, 2. aktualizované a rozšířené vydání, ComputerPress, 2005, ISBN 80-251-0509-1. *V této knize naleznete podrobné vysvětlení, pro začátečníka mnohdy překvapivého chování překladače jazyka C++.*
2. Alexandrescu A.: **Moderní programování v C++ Šablony, generické komponenty a návrhové vzory**, ComputerPress 2004, ISBN 80-251-0370-6. *Pro ty z Vás, kteří se hodlají dozvědět něco o trendech ve využití jazyka C++, nabízí tato kniha rozšiřující informace. Pozor, nejedná se přímo o učebnici jazyka C++.*
3. Koenig A., Moo B.E.: **Rozumíme C++**, ComputerPress, 2003, ISBN 80-7226-656-X. *Alternativní učebnice jazyka C++ s odlišnou metodou výkladu než předchozí doporučené. Výuka je vedena přes využití knihoven jazyka C++, okamžitý nástup objektového programování atd.*

Ostatní literatura o C++

1. Šaloun, P.: Programovací jazyk C++ pro zelenáče, Neocortex Praha, 2005, ISBN 80-86330-18-4.
2. Richta, K., Šaloun, P.: Programovací jazyk C, skriptum ČVUT, Praha 1998
3. Šaloun, P.: Programovací jazyk C. Skriptum FEI VŠB-TU Ostrava 1994
4. Kernighan, B., Ritchie, D.: Programovací jazyk C, Alfa Bratislava, 1988
5. Herout, P., Rudolf, V., Šmrha, P.: ABC programátora v jazyce C, nakladatelství KOPP, České Budějovice, 1992
6. Vondrák, I., Šaloun, P.: Objektově orientované programování, skriptum VŠB Ostrava, 1994
7. Horstmann, C. S.: Vyšší škola objektového návrhu v C++. Science, Veletiny 1997
8. Večerka, A.: Jazyk C++: popis jazyka s příklady, skriptum UP Olomouc, Olomouc 1996, ISBN 80-7067-658-2

Ostatní učební materiály

Prezentace z přednášek

- Prezentace z přednášek **nejsou** studijním materiálem.
- Slouží přednášejícímu jako osnova jeho výkladu a studentům jako přehled probrané látky.
- Pročtení těchto několika prezentací není možné považovat za dostatečnou přípravu – k tomu slouží studijní literatura a hlavně vlastní, samostatné programování.

Další materiály budou případně doplněny na webových stránkách předmětu.

Děkuji za pozornost

Lineární datové struktury

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Datové struktury

- Algoritmy (počítačové programy) pracují s daty.
- Práce s daty obnáší manipulaci s daty – čtení, zápis, vyhledávání, agregace atd.
- Cílem je efektivní manipulace s daty.
- Mezi daty existují strukturální vazby – například balíček karet, první karta, poslední karta, předchozí karta, počet karet.
- **Datová struktura** – způsob uložení dat v počítači a možnosti manipulace s těmito daty.
- Specializace datových struktur na různé úlohy.
- Datové struktury jsou obsaženy v takřka každém programu.
- Efektivní datová struktura je obvykle základem pro efektivního algoritmu.

Abstraktní datové struktury (ADS)

- **Neformální definice**: Abstraktní datová struktura není závislá na vlastní implementaci a je definována pouze pomocí povolených operací na této struktuře.
- Formálně lze definovat jako matematický model.
- Nepřímá definice pomocí operací – neříkáme co to je, ale co s tím lze dělat.
- Nezajímá nás konkrétní implementace, ale rozhraní, které struktura poskytuje.
- Zjednodušují návrh algoritmů a přinášejí abstrakci do programování.
- Konkrétní implementace je označována pojmem datová struktura.
- Implementace je, v konečném důsledku, ale také důležitá.

Abstraktní datové struktury – dělení

- Počet prvků
  - neměnný – statická ADS, např. pole pevné velikosti,
  - proměnný – dynamická ADS, počet složek je proměnný, mezi operace patří vložení, odebrání prvku
- Datový typ prvků
  - homogenní – všechny prvky jsou shodného datového typu,
  - heterogenní – prvky jsou různého datového typu
- Existence jednoho, bezprostředního, následníka
  - lineární – existuje, např. pole, zásobník, fronta, seznam, ...
  - nelineární – neexistuje, např. strom, graf, ...

Abstraktní datové struktury – typičtí představitelé

Pole (Array, Vector) – obvykle dynamická velikost

Zásobník (Stack)

Fronta (Queue)

Seznam (List, Collection) – mnoho variant

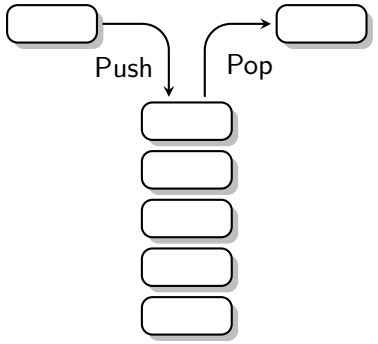
Slovník (Dictionary) – uložení dvojic (klíč, hodnota), setříděný vs. netříděný,

Množina (Set) – obdoba množiny z matematiky, setříděná vs. netříděná,

Graf (Graph) – reprezentace množiny prvků a vztahů mezi nimi, např. sociální síť

a další ...

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). Ukazatel zásobníku není nutně ukazatel ve smyslu práce s dynamicky alokovanou pamětí.
- prvně vložený prvek se nazývá **dno zásobníku** (stack bottom)

Zásobník – operace

Základní operace

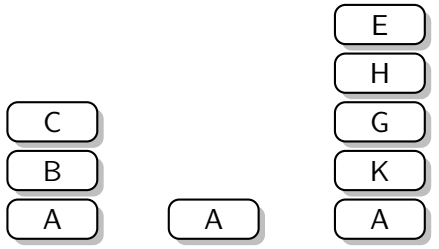
- Push – vložení prvku na vrchol zásobníku
- Pop – vyjmutí prvku z vrcholu zásobníku
- IsEmpty – test prázdnosti zásobníku
- Top – vrátí prvek z vrcholu zásobníku bez jeho vyjmutí

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




```
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).

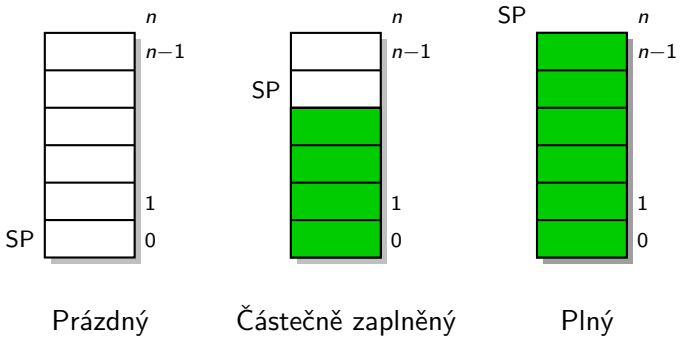
Zásobník – možnosti implementace

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



Zásobník – deklarace

```
1 const int StackSize = 100;
2
3 struct Stack
4 {
5    char Items[StackSize];
6
7    int StackPointer;
8 };
```

Zásobník – implementace

```
1 void Init(Stack& S)
2 {
3    S.StackPointer = 0;
4 }
5
6 void Push(Stack& S, const char X)
7 {
8    S.Items[S.StackPointer] = X;
9    S.StackPointer += 1;
10 }
11
12 char Pop(Stack& S)
13 {
14    S.StackPointer -= 1;
15    return S.Items[S.StackPointer];
16 }
17
18 char Top(const Stack& S)
19 {
```


Zásobník – implementace (pokrač.)

```
20    return S.Items[S.StackPointer - 1];
21 }
22
23 bool IsEmpty(const Stack& S)
24 {
25    return S.StackPointer == 0;
26 }
27
28 bool IsFull(const Stack& S)
29 {
30    return S.StackPointer == StackSize;
31 }
32
33 void Clear(Stack& S)
34 {
35    S.StackPointer = 0;
36 }
```


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 `CheckParenthesesParity`, která bude vracet `true` pokud je výraz správně uzávorkován, jinak bude vracet `false`.

Ukázka

Výraz `Expr`	Výsledek
`()`	`true`
`)(`	`false`
`(()())`	`true`
`[()()]`	`true`
`{()()()[()()]}`	`true`
`[()()`	`false`
`[(])()]`	`false`
`[[()]`	`false`

Zásobník – testování parity závorek (pokrač.)

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, levé závorka byla zpracována jako první; 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;  // nesprávná pravá závorka
11                 end
12             end
13             else
14                 return false;          // málo levých závorek
15             end
16         end
17     end
18 end
19 return IsEmpty (S);  // nesmí zůstat žádná levá závorka
```


Zásobník – testování parity závorek

```
1  bool CheckParenthesesParity1(string Expr)
2  {
3     Stack S;
4     Init(S);
5     for (size_t i = 0; i < Expr.length(); i++)
6     {
7        switch (Expr[i])
8        {
9        case '(':
10       case '[':
11       case '{':
12          Push(S, Expr[i]);
13          break;
14       case ')':
15          if (!IsEmpty(S))
16          {
17             char ch = Pop(S);
18             if (ch != '(')
19                return false;
```


Zásobník – testování parity závorek (pokrač.)

```
20          }
21          else
22             return false;
23          break;
24       case ']':
25          if (!IsEmpty(S))
26          {
27             char ch = Pop(S);
28             if (ch != '[')
29                return false;
30          }
31          else
32             return false;
33          break;
34       case '}':
35          if (!IsEmpty(S))
36          {
37             char ch = Pop(S);
38             if (ch != '{')
```

Zásobník – testování parity závorek (pokrač.)

```
39              return false;
40          }
41          else
42             return false;
43          break;
44       }
45    }
46    return IsEmpty(S);
47 }
```


Zásobník – testování parity závorek

Opakování kódu

```
15 if (!IsEmpty(S))
16 {
17    char ch = Pop(S);
18    if (ch != '(')
19       return false;
20 }
21 else
22    return false;
```

Řešení

Do zásobníku neukládat nalezenou levou závorku, ale **očekávanou** pravou závorku.

Zásobník – testování parity závorek

```
1 bool CheckParenthesesParity2(string Expr)
2 {
3    Stack S;
4    Init(S);
5    for(size_t i = 0; i < Expr.length(); i++)
6    {
7       switch (Expr[i])
8       {
9       case '(':
10         Push(S, ')');
11         break;
12      case '[':
13         Push(S, ']');
14         break;
15      case '{':
16         Push(S, '}');
17         break;
18      case ')':
19      case ']':
```


Zásobník – testování parity závorek (pokrač.)

```
20      case '}':
21         if (!IsEmpty(S))
22         {
23            char ch = Pop(S);
24            if (ch != Expr[i])
25               return false;
26         }
27         else
28            return false;
29         break;
30      }
31   }
32   return IsEmpty(S);
33 }
```

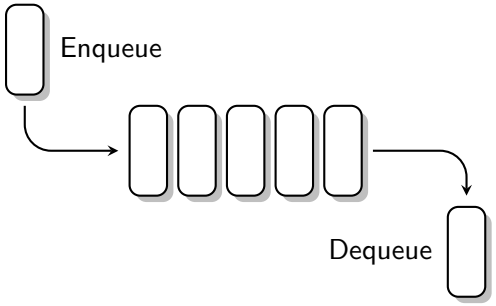
Zásobník – otázky implementace

- Jak mít v kódu (jedné funkci) více zásobníků?
- Jak zabránit, aby mohl kdokoliv v zásobníku něco měnit? Ochrana vnitřních dat.
- Lze implementovat zásobník, který by se dynamicky zvětšoval?
- Nešlo by vyrobit zásobník jako „součástku“? Třeba udělat z něj datový typ s definovanými operacemi?
- Potřebuji mít v programu zásobník čísel a znaků. To ho musím naimplementovat dvakrát? Nešlo by mít zásobník jako šablonu?

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 $\Rightarrow$ 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 ...

Fronta – možnosti implementace

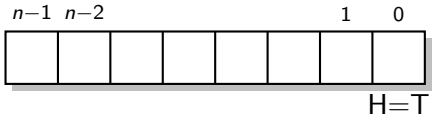
Pomocí spojových struktur – dynamická alokace paměti, pointery

Pomocí pole

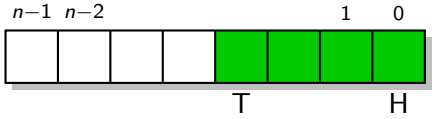
- hlava a ocas fronty jsou indexy do pole,
- jednoduché přidávání a odebírání – jak hlava, tak ocas se posunují dozadu.

Fronta – implementace v poli

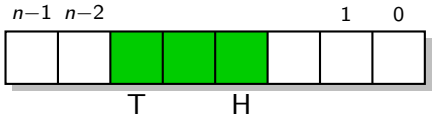
Prázdná fronta



Vloženy 4 prvky

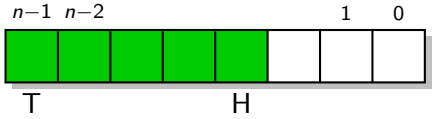


Vyjmuty 3 prvky, vloženy 2 prvky



Fronta – implementace v poli (pokrač.)

Vloženy 2 prvky



Výsledný stav:

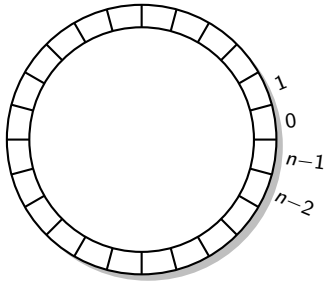
- do fronty již nelze přidat další prvky,
- ale současně fronta není plná.

Možná řešení:

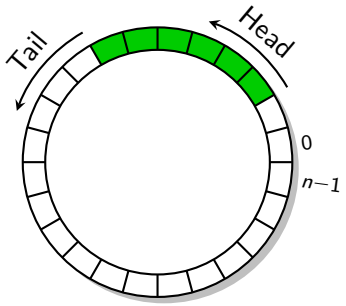
- přesouvat prvky na začátek pole – vyjmutí prvku již nebude operace s časovou složitostí $O(1)$ ale $O(n)$, bude docházet k masivnímu kopírování dat,

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 – deklarace

```
1  const int CircularBufferSize = 5;
2
3  const int MaxQueueSize = CircularBufferSize - 1;
4
5  struct Queue
6  {
7     int Items[CircularBufferSize];
8
9     int Head;
10
11    int Tail;
12 };
```


Fronta – implementace

```
1  void Init(Queue& Q)
2  {
3     Q.Head = 0;
4     Q.Tail = 0;
5  }
6
7  void Clear(Queue& Q)
8  {
9     Q.Head = 0;
10    Q.Tail = 0;
11 }
12
13 void Enqueue(Queue& Q, const int X)
14 {
15    Q.Items[Q.Tail] = X;
16    Q.Tail = (Q.Tail + 1) % CircularBufferSize;
17 }
18
19 int Dequeue(Queue& Q)
```


Fronta – implementace (pokrač.)

```
20 {
21    int x = Q.Items[Q.Head];
22    Q.Head = (Q.Head + 1) % CircularBufferSize;
23    return x;
24 }
25
26 int Peek(const Queue& Q)
27 {
28    return Q.Items[Q.Head];
29 }
30
31 bool IsEmpty(const Queue& Q)
32 {
33    return Q.Head == Q.Tail;
34 }
35
36 bool IsFull(const Queue& Q)
37 {
38    return Q.Head == (Q.Tail + 1) % CircularBufferSize;
```


Fronta – implementace (pokrač.)

```
39 }
```

Kontrolní otázky

1. Vysvětlete co znamená, že zásobník představuje paměť typu LIFO.
2. Co je to vrchol zásobníku? Co je to dno zásobníku?
3. Jaká je teoreticky kapacita zásobníku?
4. Co je to přetečení zásobníku? Co je to podtečení zásobníku?
5. Jak se typicky nazývá operace vložení prvku do zásobníku? Popište tuto operaci.
6. Jak se typicky nazývá operace vyjmutí prvku ze zásobníku? Popište tuto operaci.
7. Závisí složitost operací na počtu prvků v zásobníku? Ano či ne?
8. Uveďte příklady použití zásobníku.
9. Do zásobníku vložím prvky v pořadí A, B, C, D. Potom je budu ze zásobníku vyjímat. Jakou dostanu posloupnost?
10. Vysvětlete co znamená, že fronta představuje paměť typu FIFO.

Kontrolní otázky (pokrač.)

11. Co je to hlava fronty? Co je to ocas fronty?
12. Závisí složitost operací na počtu prvků ve frontě? Ano či ne?
13. Co je to přetečení fronty? Co je to podtečení fronty?
14. Do fronty vložím prvky v pořadí A, B, C, D. Potom je budu z fronty vyjímat. Jakou dostanu posloupnost?
15. Jak se typicky nazývá operace vložení prvku do fronty? Popište tuto operaci.
16. Jak se typicky nazývá operace vyjmutí prvku ze fronty? Popište tuto operaci.
17. Uveďte příklady použití fronty.
18. Popište hlavní problémy při implementaci fronty v poli. Jak je lze řešit pomocí kruhové reprezentace fronty?
19. Jaký je rozdíl mezi frontou, zásobníkem a seznamem? Zaměřte se na operace vkládání a vyjímání prvků z těchto struktur.

Kontrolní otázky (pokrač.)

20. Co je to hlava seznamu, ocas seznamu?
21. Rozdíl mezi jednosměrným, obousměrným a kruhovým seznamem.
22. Popište algoritmus vyhledávání daného prvku v seznamu?
23. Lze využít algoritmus půlení intervalu pro vyhledávání v seznamu? Jaký je tu hlavní problém?
24. Co je to garbage collector?
25. Co je to iterátor?

Děkuji za pozornost

Dynamická alokace paměti

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Podobenství o žlutém lístečku a kastrolu s gulášem

Pravidla hry:

- žlutý nalepovací lísteček – lísteček je volný nebo je na něm napsáno, kde je guláš nebo že guláš není
- psací stůl – lísteček je nalepen na vašem psacím stole
- kastrolek s gulášem v lednici – obsahuje lahodný pokrm, který lze konzumovat
- lednice – na základě lístečku vydá guláš, bez lístečku není guláš, lednici nelze prohledávat

Hra samotná – maminka uvaří guláš, uschová jej do lednice a synáčkovi napíše na lísteček, kde se guláš ukrývá.

Podobenství o žlutém lístečku a kastrolu s gulášem

Co můžeme říci o žlutém lístečku:

- žlutý lísteček určitě není guláš samotný
- žlutý lísteček udává **nepřímo** polohu guláše
- žlutý lísteček existuje sám o sobě, zabírá jakési malé množství paměti
- žlutý lísteček není v lednici, je jinde
- obsah žlutého lístečku – prázdný, „Guláš je v lednici“ nebo „Guláš není!“
- obsah žlutého lístečku si můžu opsat na jiný – dva lístečky odkazují na totožný guláš
- na jiný oranžový lísteček si můžu napsat kde leží žlutý lísteček
- na zelený lísteček si můžu napsat, kde leží oranžový lísteček

Podobenství o žlutém lístečku a kastrolu s gulášem

Lednice a kastrol s gulášem:

- na základě žlutého lístečku lednice vydá guláš, s gulášem lze manipulovat, například přiostřit; co se stane když si guláš vyzvednu se zeleným lístečkem?
- vydaný guláš můžu sníst; co když přijdu se zeleným lístečkem
- co když žlutý lísteček zahodím? guláš zůstane v lednici navěky
- na nový lísteček si můžu napsat že v lednici mám nový kastrol, na základě žlutého lístečku si vyzvednu kastrol s gulášem a naleji ho i do nového kastrolu a mám dva exempláře guláše (Tady dochází k porušení zákona zachování hmoty – vytváříme guláš z ničeho)

Podobenství o žlutém lístečku a kastrolu s gulášem

Informatický výklad podobenství:

- psací stůl – paměťový prostor, datový segment, místo pro uložení proměnných vzniklých v době kompilace
- lednice – paměťový prostor, halda, prostor pro dynamickou alokaci paměti
- kastrol s gulášem – dynamicky alokovaná data
- žlutý lísteček – ukazatel nebo-li pointer. Smyslem pointeru je **nepřímá adresace**. Pointer není objektem samotným (objekt ne nutně ve smyslu OOP), ale je informací, kde se objekt nalézá.

Podobenství o žlutém lístečku a kastrolu s gulášem

Možné stavy žlutého lístečku:

- čistý lísteček – neinicializovaný pointer, pointer se teď chová jako bežná proměnná
- „Guláš je v lednici“ – platný odkaz na guláš, lze provést dereferenci
- „Guláš není!“ – zvláštní, vyhrazená, hodnota. Pointer ukazuje „nikam“. Nelze zaměňovat s čistým lístečkem. Pointer nikam lze testovat ale nelze provést dereferenci.
-

```
1 DatovyTyp *ZlutyListecek;
2 int *p;
3 int *q = NULL;
4 int *r = nullptr;
```


Podobenství o žlutém lístečku a kastrolu s gulášem

Uvaření nového guláše:

- uvaření guláše lze chápat jako dynamickou alokaci objektu v paměti
- alokaci provedeme operátorem `new`

```
1 struct Kastrol
2 {
3   // gulas
4 };
5 Kastrol *ZlutyListecek = new Kastrol;
```


Podobenství o žlutém lístečku a kastrolu s gulášem

Manipulace s gulášem:

- na základě žlutého lístečku lze získat z lednice kastrol s gulášem a s tímto gulášem manipulovat
- operace získání guláše se nazývá dereference, „odpointrování“

```
1 Kastrol K = *ZlutyListecek;
2 K += LzicePapriky;
```

nebo přímo

```
1 *ZlutyListecek += LzicePapriky;
```

Dereference neinicializovaného pointeru nebo `nullptr`

Neexistující objekty nelze dereferencovat! Fatální chyba.

Podobenství o žlutém lístečku a kastrolu s gulášem

Likvidace guláše:

- likvidaci guláše (snězení, vylití) lze chápat jako delokaci objektu v paměti
- delokaci provedeme pomocí operátoru `delete`

```
1 delete ZlutyListecek;
```

Memory leak

Každý dynamicky alokovaný objekt musí být uvolněn z paměti! Jinak dojde k úniku paměti. Nepoužívané objekty za vás nikdo neuklízí.

Podobenství o žlutém lístečku a kastrolu s gulášem

Operace s pointery:

- **derefence** – na základě žlutého lístečku dostanu kastrol s gulášem, „odpointrování“
- **reference** – získání umístění objektu, poloha guláše v lednici, výsledek lze zapsat na žlutý lísteček
- reference a dereference jsou inverzní operace

```
1 struct Kastrol
2 {
3   // nejaka data
4 };
5 Kastrol JinyHotovyGulas;
6 Kastrol *ZlutyListecek = &JinyHotovyGulas;
```


Podobenství o žlutém lístečku a kastrolu s gulášem

Operace s pointery:

```
1 Kastrol *ZlutyListecek;
2 Kastrol *ZelenyListecek;
```

- přiřazení pointerů `ZelenyListecek = ZlutyListecek`, ve výsledku oba lístečky ukazují na tentýž kastrol, nebo na nic, nebo mají shodnou neinicializovanou hodnotu
- přiřazení objektů na které pointery ukazují `*ZelenyListecek = *ZlutyListecek`, ve výsledku se obsah guláše přelil do dašího kastrolu

Co pointer není

- Pointer není adresa v paměti. Ano pointer musí nutně skončit jako nějaká binární hodnota, kterou lze interpretovat jako číslo.
- Nelze si ale představovat, že na daném bajtu v paměti bychom nějakým přístojem našli binární ekvivalent odkazovaných dat.
- `NULL` není nula ani jiné přirozené číslo. Možná konverze na `int` je vražedným dědictvím ANSI C. Fuj! Nebrat!
- Jazyk C++ nově (od verze C++ 11) obsahuje `nullptr`.

Podobenství o kabátu v šatně na plese

Situace:

- Vesnický hasičský, sokolský či jiný ples.
- Šatna, kde odevzdáváte kabáty je vybavena pojízdnými věšáky.
- Odevzdáte kabát a dostanete papírový lísteček s číslem například 118.
- Po plese si na základě lístečku vyzvednete kabát.
- Jestliže se na jeden věšák vejde 20 kabátů a začaly se věšáky plnit zleva, tak by šlo usuzovat, že váš kabát 118 je na šestém věšáku osmnáctý v pořadí.
- Vzhledem k tomu, že si šatnářky mohou věšáky přerovnávat, tak se nedá říci, kde se věšák v vašim kabátem v šatně nachází.
- Ale víme, že na základě lístečku kabát dostanu. Zbytek je záležitost šatnářky.
- Šatna plní roli paměti, šatnářka je memory manager, heap manager.
- Přesuny věšáků – stránkování paměti, virtuální paměť atd.

Konstantní pointery

- Co lze označit za konstantní: polohu a obsah.
- Celkem mohou nastat 4 možnosti:
- `int* p;` – vše proměnné
- `const int* p;` – konstantní obsah
- `int* const p;` – konstantní poloha
- `const int* const p;` – konstantní obsah i poloha.

Dynamická alokace pole v C++

- Identifikátor pole je pointer na první prvek tohoto pole.
- Jednorozměrné pole

```
1 int *A = new int[5];
2 delete []A;
```

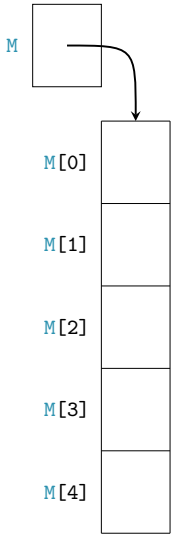
- Dvourozměrné pole

```
1 double **M = new double*[R];
2 for (int i = 0; i < R; i++)
3   M[i] = new double[C];
4
5 for (int i = 0; i < R; i++)
6   delete[]M[i];
7 delete []M;
```

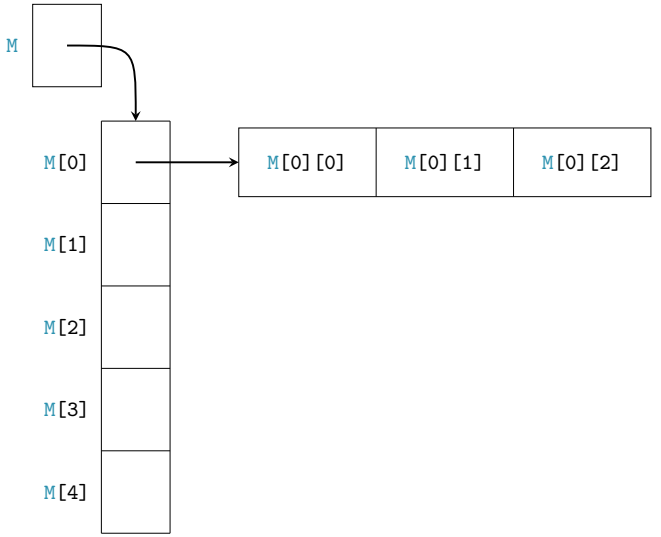

Dynamická alokace dvourozměrného pole v C++



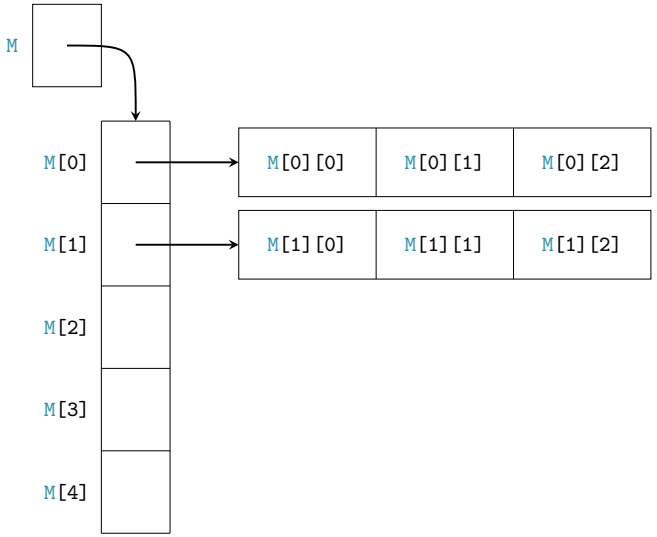
Dynamická alokace dvourozměrného pole v C++ (pokrač.)



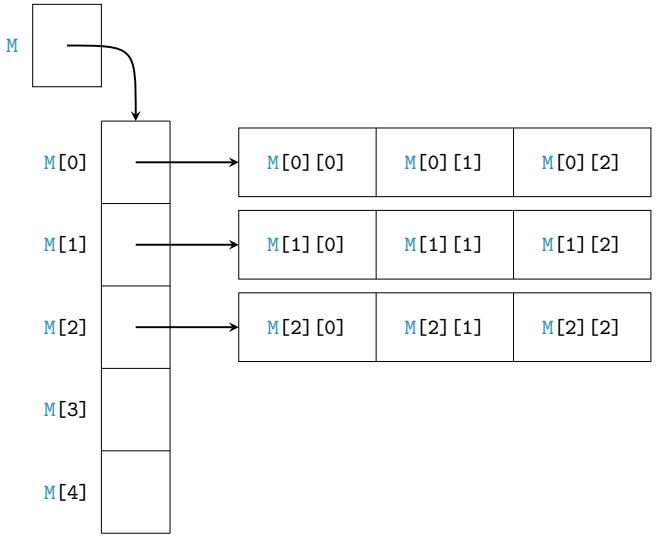
Dynamická alokace dvourozměrného pole v C++ (pokrač.)



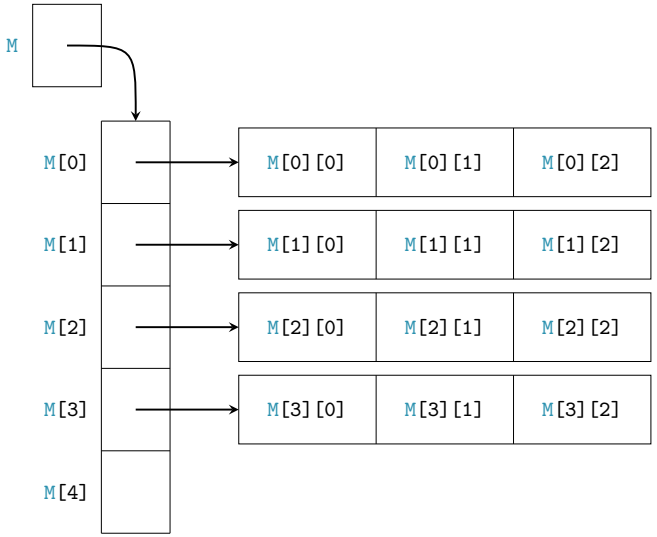
Dynamická alokace dvourozměrného pole v C++ (pokrač.)



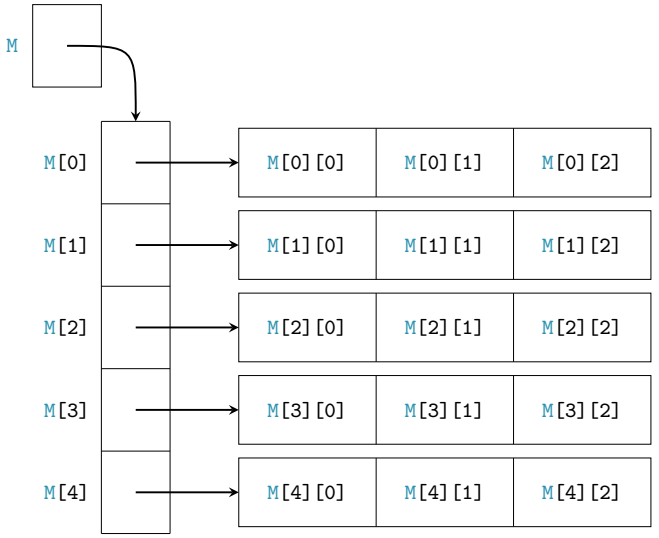
Dynamická alokace dvourozměrného pole v C++ (pokrač.)



Dynamická alokace dvourozměrného pole v C++ (pokrač.)



Dynamická alokace dvourozměrného pole v C++ (pokrač.)



Možnosti předávání parametrů funkcím

Rozlišujeme tyto způsoby předávání parametrů:

- hodnotou – vznik lokální kopie parametru
- pointerem – předáváno, kopírováno, umístění objektu
- odkazem – předáváno, kopírováno, umístění objektu. Obdoba volání pomocí pointeru, pointer samotný je před programátorem utajen kompilátorem.

Předávání polí:

- Pole jsou vždy předávána jako pointer.
- `void F(int A[])` nebo `void F(int* A)`
- Pozor na chování operátoru `sizeof`

Možnosti předávání parametrů funkcím (pokrač.)

```
1 void F(int* B)
2 {
3    cout << sizeof(B);
4 }
5
6 int A[5];
7 cout << sizeof(A);
8 F(A);
```

Pointerová artimetika

```
1 void CopyArray1(int* SourceArray, int* DestinationArray, const int N)
2 {
3    for (int i = 0; i < N; i++)
4    {
5       DestinationArray[i] = SourceArray[i];
6    }
7 }
```


Pointerová artimetika

```
1 void CopyArray2(int* SourceArray, int* DestinationArray, const int N)
2 {
3    int* SPtr = SourceArray;
4    int* DPtr = DestinationArray;
5    int* EndPtr = SourceArray + N;
6    while (SPtr != EndPtr)
7    {
8       *DPtr = *SPtr;
9       SPtr += 1;
10      DPtr += 1;
11   }
12 }
```


Pointerová artimetika

```
1 void CopyArray2a(int* SourceArray, int* DestinationArray, const int N)
2 {
3    for (int *SPtr = SourceArray, *DPtr = DestinationArray; SPtr !=
         SourceArray + N; *DPtr++ = *SPtr++);
4 }
```


Kontrolní otázky

1. Co je to pointer? Jaký je význam dat obsažených v pointeru? Nápověda: zaměřte se na vysvětlení pojmu nepřímého adresování.
2. Jak deklarujeme pointer `p` na typ `int`?
3. Jak se nazývá pointer ukazující „nikam“?
4. Co je to operátor reference? Uveďte příklad použití.
5. Co je to operátor dereference? Uveďte příklad použití.
6. Předpokládejme, že máme dánu proměnnou `int a = 8;` a pointer `int *p;`. Jakým způsobem zařídíte, aby pointer `p` ukazoval na proměnnou `a`?
7. Předpokládejme, že máme deklarovány dva pointery `int* p;` a `int* q;`. Co se stane po vyhodnocení výrazu `p = q;`?
8. Předpokládejme, že máme deklarovány dva pointery `int* p;` a `int* q;`. Co se stane po vyhodnocení výrazu `*p = *q;`? Jaké zde můžou nastat problémy?

Kontrolní otázky (pokrač.)

9. Deklarujte pointer `d` na typ `double`. Proveďte alokaci paměti a dealokaci paměti. Jaké operátory na to použijete?
10. Máte dynamicky alokovat pole `a` s prvky typu `int`, které bude mít 20 prvků. Jak to provedete? Pomocí jakého operátoru?
11. Je dáno dynamicky alokované pole `a` s prvky typu `int`, které má 20 prvků. Máte toto pole dealokovat. Jak to provedete? Pomocí jakého operátoru?
12. Máte za úkol dynamicky alokovat dvourozměrné pole `a`, které bude bude mít `R` řádků a `S` sloupců. Prvky budou typu `int`. Jak bude vypadat deklarace pointeru na takové pole? Jak se provede jeho alokace v paměti? Lze to provést jednorázovou alokací?
13. Máte za úkol uvolnit z paměti, dealokovat, dvourozměrné pole `a`, které má `R` řádků a `S` sloupců. Prvky jsou typu `int`. Jak to provedete? Lze to provést jednorázovou dealokací?

Kontrolní otázky (pokrač.)

14. Máte pointer daný následující deklarací `int* p;`. Co je na tomto pointeru konstantního a co proměnného?
15. Máte pointer daný následující deklarací `const int* p;`. Co je na tomto pointeru konstantního a co proměnného?
16. Máte pointer daný následující deklarací `int* const p;`. Co je na tomto pointeru konstantního a co proměnného?
17. Máte pointer daný následující deklarací `const int* const p;`. Co je na tomto pointeru konstantního a co proměnného?
18. Popište tři způsoby předávání parametru do funkce.
19. Do funkce `F` předáváte pole pomocí pointeru. Ve funkci `F` na takto předané pole (na parametr z hlavičky funkce) aplikujete operátor `sizeof`? Jaký výsledek dostanete?

Kontrolní otázky (pokrač.)

20. Jaké operace jsou povoleny provádět s pointery v pointerové aritmetice? Za jakého předpokladu mají tyto operace smysl? Nápověda: Mají tyto operace smysl pokud pointery ukazují na libovolné místo v paměti? Nebo musí být s něčím svázány?
21. Jak je možné chápat pole z pohledu pointeru? Co vlastně z tohoto pohledu představuje identifikátor pole?
22. Máte dáno pole `int a[10];` a ukazatel `int* p = a;`. Čemu odpovídá výraz `*(p+3)`?
23. Máte dáno pole `int a[10];` a ukazatel `int* p = a;`. Kam bude ukazovat pointer `p` po vyhodnocení výrazu `p+=2;`?
24. Máte dva pointery `int *p, *q;`. Jaký je smysl výrazu `p == q`?
25. Máte dáno pole `int a[10];` a dva ukazatele `int *p, *q;`. Oba tyto ukazatele ukazují na některý prvek v poli `a`;. Jaký je smysl výrazů `p-q` a `*p-*q`?

Děkuji za pozornost

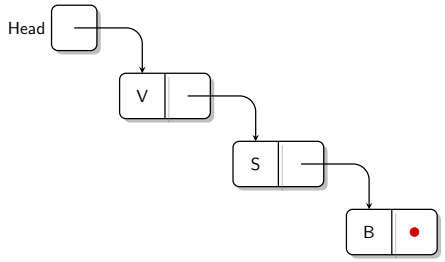
Spojová implementace lineárních datových struktur

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Seznam (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**.

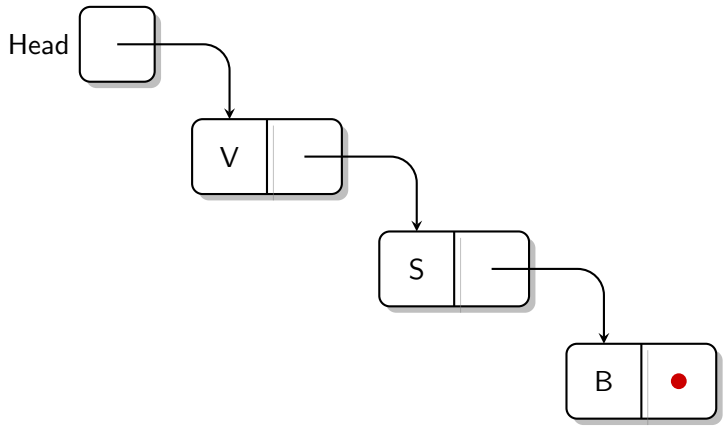
Seznam – jednosměrný seznam

- 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**.

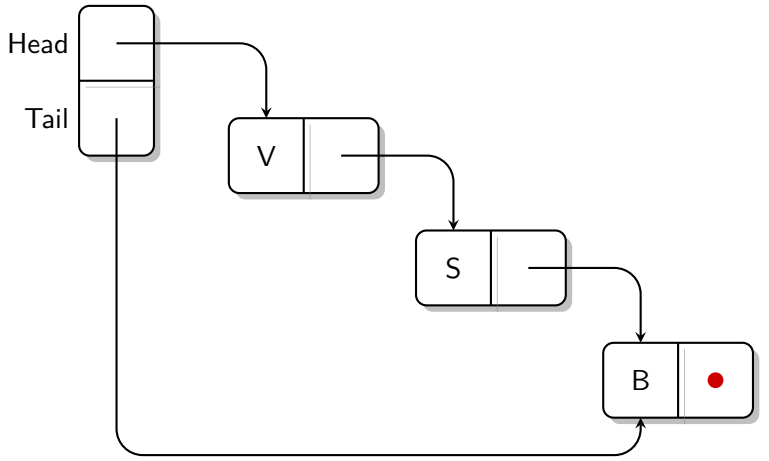
Seznam, varianty

- jednosměrný seznam – nejjednodušší varianta, odkaz pouze na následníka,
- obousměrný seznam – položka obsahuje odkaz na předchůdce i následníka,
- kruhový seznam – hlava a ocas seznamu splývají.

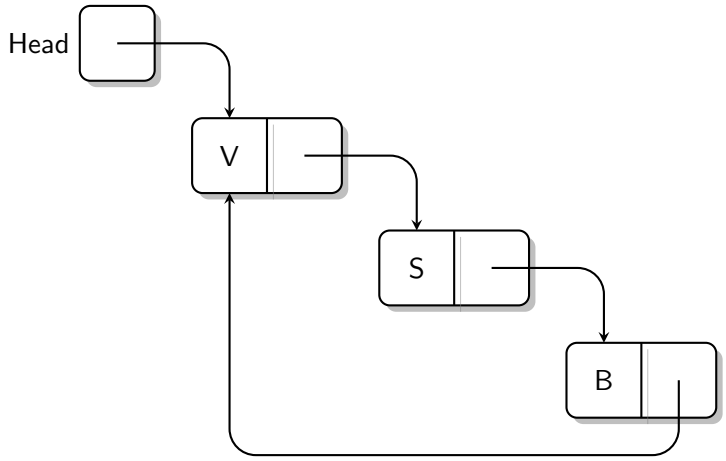
Seznam – jednosměrný, jen hlava seznamu



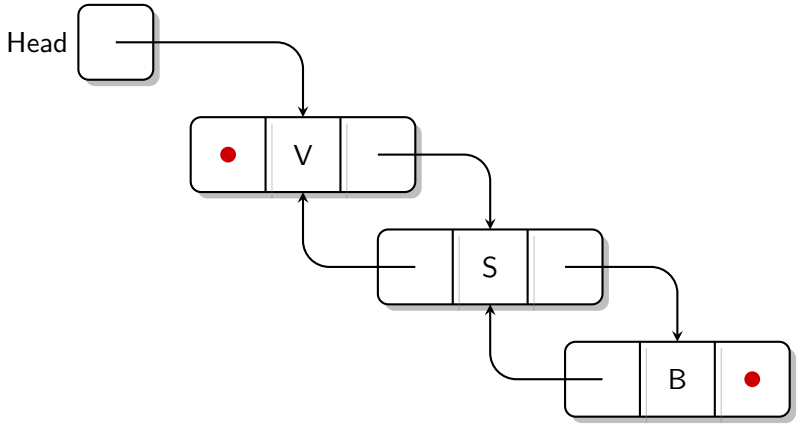
Seznam – jednosměrný, hlava i ocas seznamu



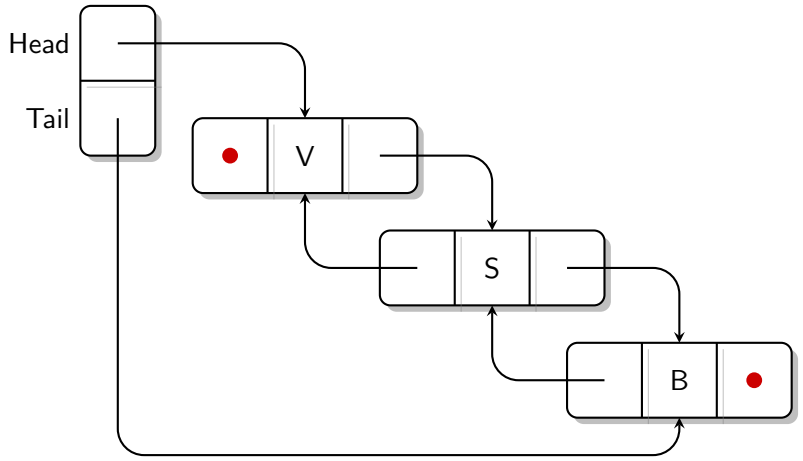
Seznam – jednosměrný, kruhový



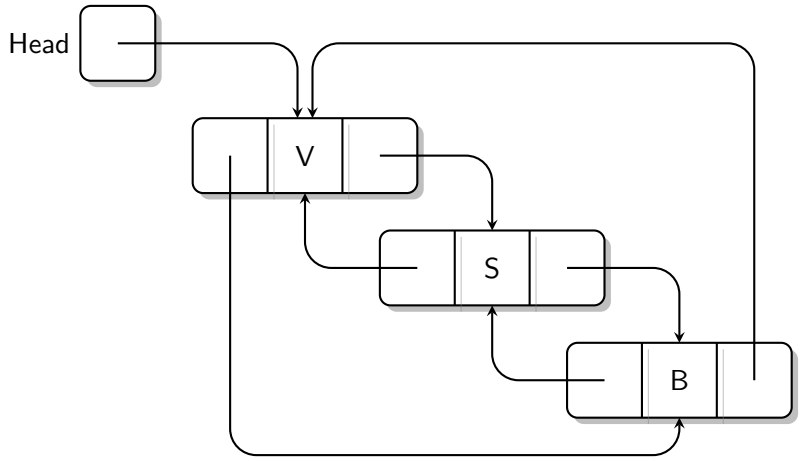
Seznam – obousměrný, jen hlava seznamu



Seznam – obousměrný, hlava i ocas seznamu



Seznam – obousměrný, kruhový



Seznam – použití

- obecná datová struktura,
- dynamicky se měnící kolekce prvků,
- čili kdekoliv – například velice primitivní databáze,
- otázkou ale je složitost jednotlivých operací se seznamem – třídění, vyhledávání prvku atd.

Seznam – možnosti implementace

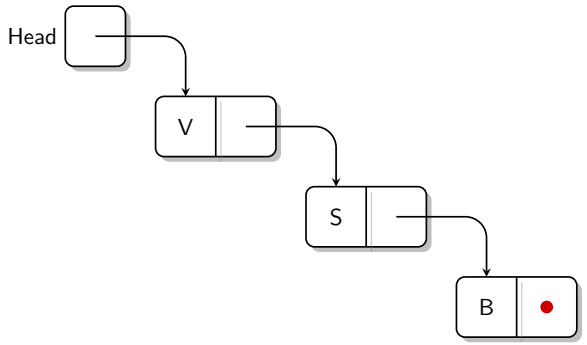
- pomocí pole nebo
- pomocí spojových struktur

Seznam – implementace pomocí pole

- pole délky n – omezená kapacita, ale rychlá manipulace,
- odkazy na předchůdce resp. následníka jsou reprezentovány indexy do pole – čísly `int`,
- uložení položek seznamu v poli nemusí odpovídat logickému uspořádání položek v seznamu,
- správa paměti:
  - evidence volného prostoru v poli,
  - přidělování prostoru z evidence pro nové položky při vkládání,
  - navracení prostoru do evidence při mazání položky.

Logická struktura vs. uložení dat

Logická struktura



Paměťová struktura

Head: 8
Free: 3

	0	1	2	3	4	5	6	7	8	9
Value			**S**			**B**			**V**	
Next	1	4	5	9	null	null	7	0	2	6

Garbagge Collector (GC)

- mechanismus pro evidenci volného prostoru v poli,
- doslova „sběr odpadu“ – sběr nepoužívané paměti,
- index **Free** ukazuje na první volnou položku,
- v nepoužívaných položkách seznamu je využita složka **Next**, která odkazuje na další volnou položku,
- při alokaci položky si vezmeme první volnou a **Free** posuneme přes **Next** na další volnou položku,
- při dealokaci vložíme uvolňovanou položku před položku na kterou odkazuje **Free**,
- GC tedy pracuje na principu LIFO, chová tedy jako **zásobník** volného místa.

Seznam pomocí pole – deklarace struktur

Konstanty

```
1 const int ListSize = 5;
2
3 const int Nothing = -1;
```

Položka seznamu

```
1 struct ListItem
2 {
3    string Value;
4    int Prev;
5    int Next;
6 };
```

Vlastní seznam

```
1 struct List
2 {
3    ListItem Items[ListSize];
4    int Head;
5    int Tail;
6    int Free;
7 };
```

Seznam pomocí pole – deklarace funkcí

```
1 void Init(List& L);
2 void Init(List& L, const string& InitValue);
3 void Init(List& L, const string InitValues[], const int N);
4 void Init(List& L, const List& Other);
5
6 void Prepend(List& L, const string& NewValue);
7 void Append(List& L, const string& NewValue);
8 void InsertBefore(List& L, const int CurrentItem, const string&
      NewValue);
9 void InsertAfter(List& L, const int CurrentItem, const string& NewValue
      );
10
11 void RemoveHead(List& L);
12 void RemoveTail(List& L);
13 void RemoveFirst(List& L, const string& ValueToRemove);
14 void RemoveLast(List& L, const string& ValueToRemove);
15 void RemoveAll(List& L, const string& ValueToRemove);
16 void Remove(List& L, const int ItemToRemove);
17 void Clear(List& L);
```


Seznam pomocí pole – deklarace funkcí (pokrač.)

```
18
19 int Search(const List& L, const string& Value);
20 int ReverseSearch(const List& L, const string& Value);
21
22 bool Contains(const List& L, const string& Value);
23
24 bool IsEmpty(const List& L);
25 int Count(const List& L);
26
27 void Report(const List& L);
28 void ReportStructure(const List& L);
29
30 void InternalCreateSingleElementList(List& L, const string& NewValue);
31 void InternalRemove(List& L, const int ItemToDelete);
32 int InternalAllocateItem(List& L);
33 void InternalFreeItem(List& L, const int Index);
```


Seznam pomocí pole – implementace

```
1 void Init(List& L)
2 {
3    Clear(L);
4 }
5
6 void Init(List& L, const string& InitValue)
7 {
8    Init(L);
9    InternalCreateSingleElementList(L, InitValue);
10 }
11
12 void Init(List& L, const string InitValues[], const int N)
13 {
14    Init(L);
15    for (int i = 0; i < N; i++)
16    {
17       Append(L, InitValues[i]);
18    }
19 }
```


Seznam pomocí pole – implementace (pokrač.)

```
20
21 void Init(List& L, const List& Other)
22 {
23    Init(L);
24    for (int p = Other.Head; p != Nothing; p = Other.Items[p].Next)
25    {
26       Append(L, Other.Items[p].Value);
27    }
28 }
29
30 void Prepend(List& L, const string& NewValue)
31 {
32    if (IsEmpty(L))
33    {
34       InternalCreateSingleElementList(L, NewValue);
35    }
36    else
37    {
38       int NewItemIndex = InternalAllocateItem(L);
```

Seznam pomocí pole – implementace (pokrač.)

```
39       L.Items[NewItemIndex].Value = NewValue;
40       L.Items[NewItemIndex].Prev = Nothing;
41       L.Items[NewItemIndex].Next = L.Head;
42       L.Items[L.Head].Prev = NewItemIndex;
43       L.Head = NewItemIndex;
44    }
45 }
46
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
54    {
55       int NewItemIndex = InternalAllocateItem(L);
56       L.Items[NewItemIndex].Value = NewValue;
57       L.Items[NewItemIndex].Prev = L.Tail;
```


Seznam pomocí pole – implementace (pokrač.)

```
58       L.Items[NewItemIndex].Next = Nothing;
59       L.Items[L.Tail].Next = NewItemIndex;
60       L.Tail = NewItemIndex;
61    }
62 }
63
64 void InsertBefore(List& L, const int CurrentItem, const string&
       NewValue)
65 {
66    if (CurrentItem != Nothing)
67    {
68       if (CurrentItem == L.Head)
69       {
70          Prepend(L, NewValue);
71       }
72       else
73       {
74          int NewItemIndex = InternalAllocateItem(L);
75          L.Items[NewItemIndex].Value = NewValue;
```


Seznam pomocí pole – implementace (pokrač.)

```
76          int P = L.Items[CurrentItem].Prev;
77          L.Items[P].Next = NewItemIndex;
78          L.Items[NewItemIndex].Prev = P;
79          L.Items[CurrentItem].Prev = NewItemIndex;
80          L.Items[NewItemIndex].Next = CurrentItem;
81       }
82    }
83 }
84
85 void InsertAfter(List& L, const int CurrentItem, const string& NewValue
       )
86 {
87    if (CurrentItem != Nothing)
88    {
89       if (CurrentItem == L.Tail)
90       {
91          Append(L, NewValue);
92       }
93       else
```


Seznam pomocí pole – implementace (pokrač.)

```
94       {
95          int NewItemIndex = InternalAllocateItem(L);
96          L.Items[NewItemIndex].Value = NewValue;
97          int N = L.Items[CurrentItem].Next;
98          L.Items[N].Prev = NewItemIndex;
99          L.Items[NewItemIndex].Next = N;
100         L.Items[CurrentItem].Next = NewItemIndex;
101         L.Items[NewItemIndex].Prev = CurrentItem;
102      }
103   }
104 }
105
106 void RemoveHead(List& L)
107 {
108   if (!IsEmpty(L))
109   {
110      InternalRemove(L, L.Head);
111   }
112 }
```

Seznam pomocí pole – implementace (pokrač.)

```
113
114 void RemoveTail(List& L)
115 {
116    if (!IsEmpty(L))
117    {
118       InternalRemove(L, L.Tail);
119    }
120 }
121
122 void RemoveFirst(List& L, const string& ValueToRemove)
123 {
124    int p = Search(L, ValueToRemove);
125    if (p != Nothing)
126    {
127       InternalRemove(L, p);
128    }
129 }
130
131 void RemoveLast(List& L, const string& ValueToRemove)
```


Seznam pomocí pole – implementace (pokrač.)

```
132 {
133    int p = ReverseSearch(L, ValueToRemove);
134    if (p != Nothing)
135    {
136       InternalRemove(L, p);
137    }
138 }
139
140 void RemoveAll(List& L, const string& ValueToRemove)
141 {
142    int p;
143    while ((p = Search(L, ValueToRemove)) != Nothing)
144    {
145       InternalRemove(L, p);
146    }
147 }
148
149 void Remove(List& L, const int ItemToRemove)
150 {
```


Seznam pomocí pole – implementace (pokrač.)

```
151    if (ItemToRemove != Nothing)
152    {
153       InternalRemove(L, ItemToRemove);
154    }
155 }
156
157 void Clear(List& L)
158 {
159    L.Head = Nothing;
160    L.Tail = Nothing;
161    for (int i = 0; i < ListSize; i++)
162    {
163       L.Items[i].Next = i + 1;
164    }
165    L.Items[ListSize - 1].Next = Nothing;
166    L.Free = 0;
167 }
168
169 int Search(const List& L, const string& Value)
```


Seznam pomocí pole – implementace (pokrač.)

```
170 {
171    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
172    {
173       if (L.Items[p].Value == Value)
174       {
175          return p;
176       }
177    }
178    return Nothing;
179 }
180
181 int ReverseSearch(const List& L, const string& Value)
182 {
183    for (int p = L.Tail; p != Nothing; p = L.Items[p].Prev)
184    {
185       if (L.Items[p].Value == Value)
186       {
187          return p;
188       }
```

Seznam pomocí pole – implementace (pokrač.)

```
189    }
190    return Nothing;
191 }
192
193 bool Contains(const List& L, const string& Value)
194 {
195    return Search(L, Value) != Nothing;
196 }
197
198 bool IsEmpty(const List& L)
199 {
200    return L.Head == Nothing;
201 }
202
203 int Count(const List& L)
204 {
205    int counter = 0;
206    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
207    {
```


Seznam pomocí pole – implementace (pokrač.)

```
208       counter += 1;
209    }
210    return counter;
211 }
212
213 void Report(const List& L)
214 {
215    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
216    {
217       cout << L.Items[p].Value << "\t";
218    }
219    cout << endl;
220 }
221
222 void ReportStructure(const List& L)
223 {
224    cout << "L.Head: " << L.Head << endl;
225    cout << "L.Tail: " << L.Tail << endl;
226    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
```


Seznam pomocí pole – implementace (pokrač.)

```
227    {
228       cout << "Item address: " << p << endl;
229       cout << "Value: " << L.Items[p].Value << endl;
230       cout << "Prev: " << L.Items[p].Prev << endl;
231       cout << "Next: " << L.Items[p].Next << endl;
232       cout << endl;
233    }
234 }
235
236 void InternalCreateSingleElementList(List& L, const string& NewValue)
237 {
238    L.Head = L.Tail = InternalAllocateItem(L);
239    L.Items[L.Head].Value = NewValue;
240    L.Items[L.Head].Prev = Nothing;
241    L.Items[L.Head].Next = Nothing;
242 }
243
244 void InternalRemove(List& L, const int ItemToDeleteIndex)
245 {
```


Seznam pomocí pole – implementace (pokrač.)

```
246    if (ItemToDeleteIndex == L.Head && ItemToDeleteIndex == L.Tail)
247    {
248       L.Head = Nothing;
249       L.Tail = Nothing;
250    }
251    else
252    {
253       if (ItemToDeleteIndex == L.Head)
254       {
255          L.Head = L.Items[L.Head].Next;
256          L.Items[L.Head].Prev = Nothing;
257       }
258       else
259       {
260          if (ItemToDeleteIndex == L.Tail)
261          {
262             L.Tail = L.Items[L.Tail].Prev;
263             L.Items[L.Tail].Next = Nothing;
264          }
```

Seznam pomocí pole – implementace (pokrač.)

```
265         else
266         {
267            int P = L.Items[ItemToDeleteIndex].Prev;
268            int N = L.Items[ItemToDeleteIndex].Next;
269            L.Items[P].Next = N;
270            L.Items[N].Prev = P;
271         }
272      }
273   }
274   InternalFreeItem(L, ItemToDeleteIndex);
275 }
276
277 int InternalAllocateItem(List& L)
278 {
279   if (L.Free != Nothing)
280   {
281      int p = L.Free;
282      L.Free = L.Items[L.Free].Next;
283      return p;
```


Seznam pomocí pole – implementace (pokrač.)

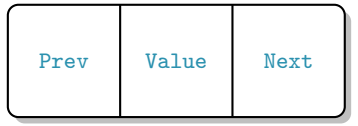
```
284   }
285   cerr << "List is full" << endl;
286   return Nothing;
287 }
288
289 void InternalFreeItem(List& L, const int Index)
290 {
291   L.Items[Index].Next = L.Free;
292   L.Free = Index;
293 }
```


Spojová implementace

- jednotlivé položky seznamu jsou samostatně alokovány heap managerem,
- vazby mezi položkami realizovány pomocí ukazatelů,
- velikost seznamu omezena dostupnou pamětí,
- nebezpečí úniku paměti (memory leak) – nutnost precizní práce s ukazateli a korektní dealokace položek.

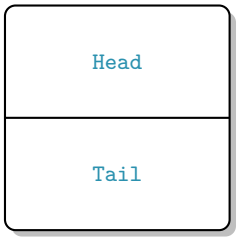
Položka seznamu

```
1 struct ListItem
2 {
3    string Value;
4    ListItem* Prev;
5    ListItem* Next;
6 };
```



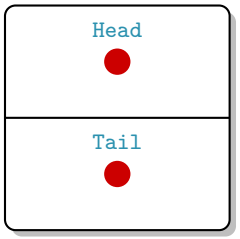
Datová reprezentace seznamu

```
1 struct List
2 {
3    ListItem* Head;
4    ListItem* Tail;
5 };
```



Inicializace prázdného seznamu

```
1 void Init(List& L)
2 {
3    L.Head = nullptr;
4    L.Tail = nullptr;
5 }
```



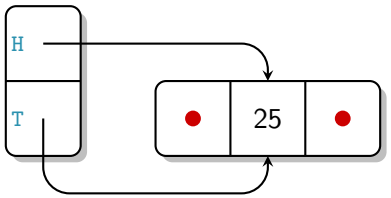
Inicializace seznamu jedním prvkem

```
7 void Init(List& L, const string& InitValue)
8 {
9    InternalCreateSingleElementList(L, InitValue);
10 }
```

Inicializaci převedeme na volání další, interní, funkce – vytvoření seznamu s jediným prvkem se totiž v kódu několikrát opakuje.

Vytvoření seznamu s jediným prvkem

```
232 void InternalCreateSingleElementList(List& L, const string& NewValue)
233 {
234    L.Head = L.Tail = new ListItem;
235    L.Head->Value = NewValue;
236    L.Head->Prev = nullptr;
237    L.Head->Next = nullptr;
238 }
```



Vložení položky do seznamu

Vložení položky do seznamu můžeme rozdělit na dvě fáze:

1. alokace nové položky v paměti a
2. zapojení nové položky do struktury seznamu – navázání vazeb mezi stávajícími položkami a novou položkou. Zde mohou nastat celkem čtyři situace, podle toho kam novou položku vkládáme:
   - 2.1 před hlavu seznamu – funkce `Prepend`,
   - 2.2 za ocas fronty – funkce `Append`,
   - 2.3 před danou položku – funkce `InsertBefore` a
   - 2.4 za danou položku – funkce `InsertAfter`.

Vložení položky do seznamu – před hlavu seznamu

Zde mohou nastat dvě situace:

1. seznam je prázdný – vytváříme seznam s jediným prvkem a
2. seznam není prázdný.

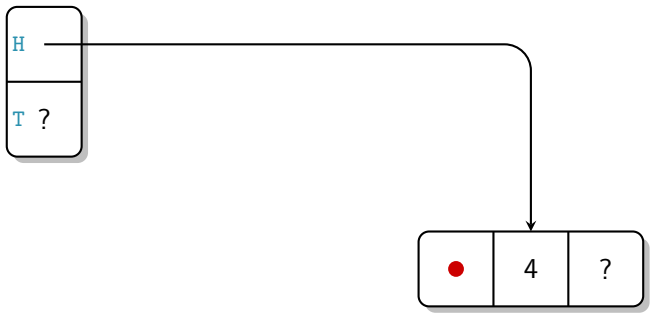
```
30 void Prepend(List& L, const string& NewValue)
31 {
32    if (IsEmpty(L))
33    {
34       InternalCreateSingleElementList(L, NewValue);
35    }
36    else
37    {
38       ListItem* NewItem = new ListItem;
39       NewItem->Value = NewValue;
40       NewItem->Prev = nullptr;
41       NewItem->Next = L.Head;
42       L.Head->Prev = NewItem;
43       L.Head = NewItem;
```


Vložení položky do seznamu – před hlavu seznamu (pokrač.)

```
44    }
45 }
```

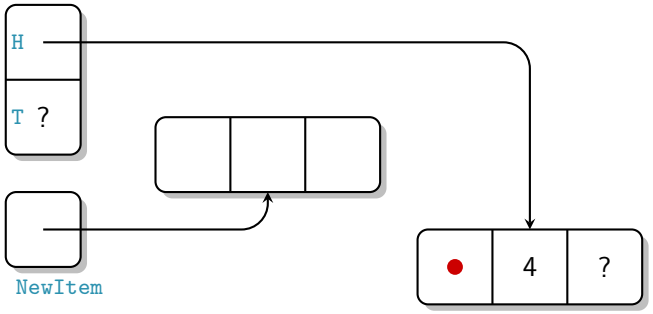

Vložení položky před hlavu seznamu – neprázdný seznam 1

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



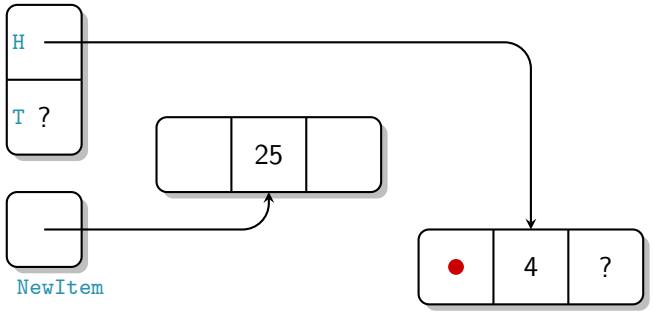
Vložení položky před hlavu seznamu – neprázdný seznam 2

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



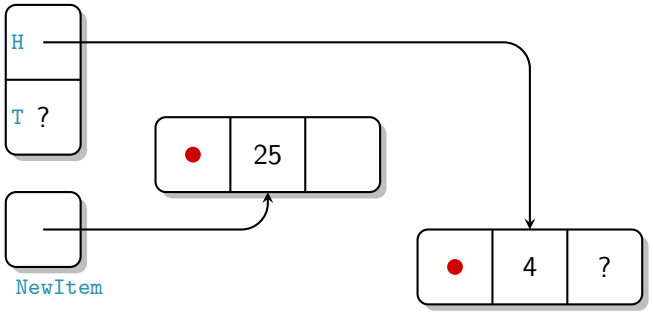
Vložení položky před hlavu seznamu – neprázdný seznam 3

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



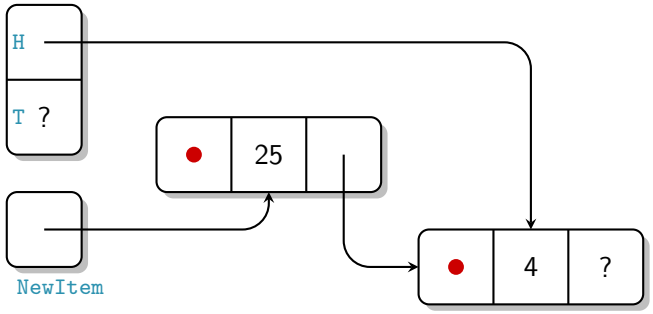
Vložení položky před hlavu seznamu – neprázdný seznam 4

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



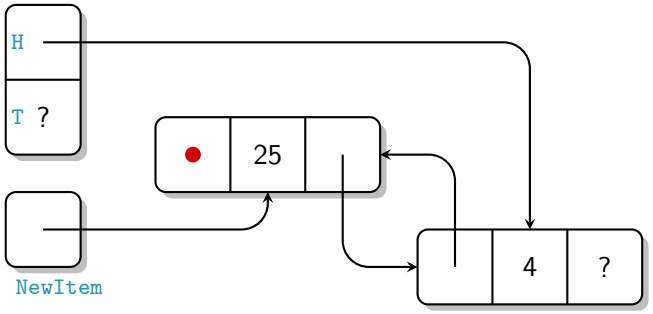
Vložení položky před hlavu seznamu – neprázdný seznam 5

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



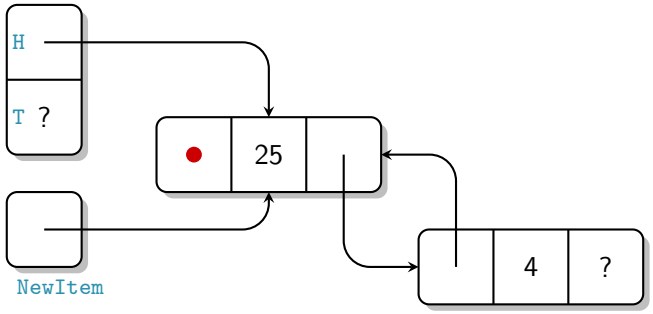
Vložení položky před hlavu seznamu – neprázdný seznam 6

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



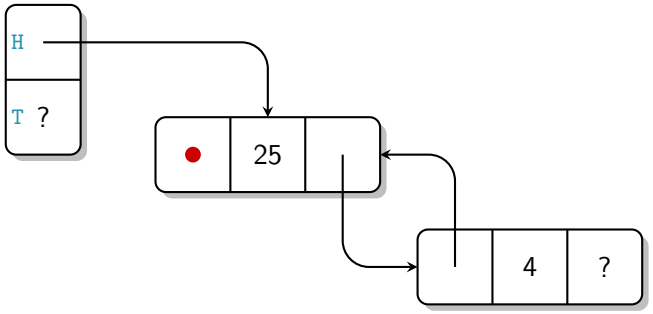
Vložení položky před hlavu seznamu – neprázdný seznam 7

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



Vložení položky před hlavu seznamu – neprázdný seznam 8

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



Vložení položky do seznamu – na konec seznamu

Zde mohou nastat dvě situace:

1. seznam je prázdný – vytváříme seznam s jediným prvkem a
2. seznam není prázdný.

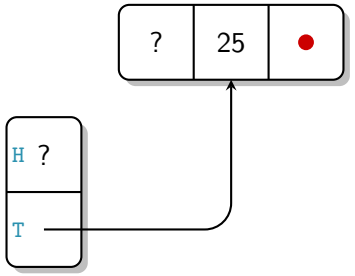
```
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
54    {
55       ListItem* NewItem = new ListItem;
56       NewItem->Value = NewValue;
57       NewItem->Next = nullptr;
58       L.Tail->Next = NewItem;
59       NewItem->Prev = L.Tail;
60       L.Tail = NewItem;
```

Vložení položky do seznamu – na konec seznamu (pokrač.)

```
61     }
62 }
```

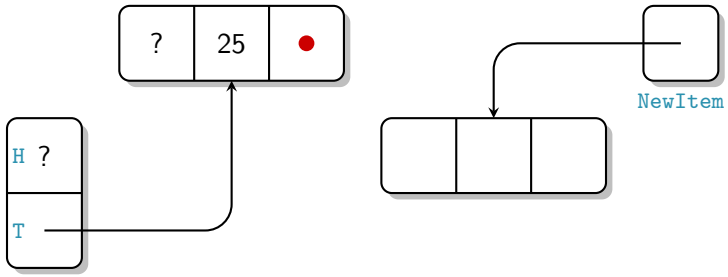

Vložení položky na konec seznamu – neprázdný seznam 1

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



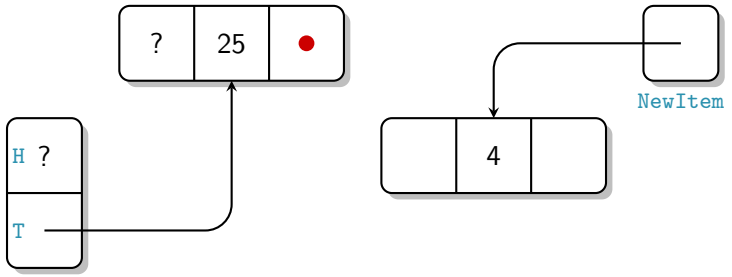
Vložení položky na konec seznamu – neprázdný seznam 2

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



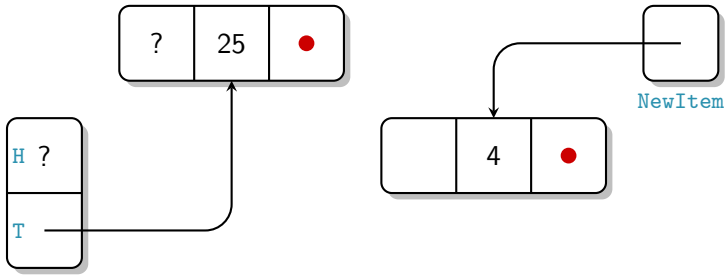
Vložení položky na konec seznamu – neprázdný seznam 3

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



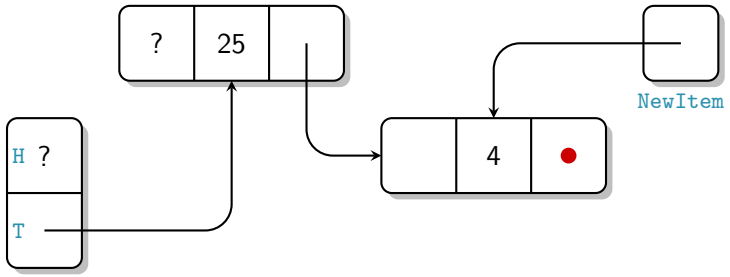
Vložení položky na konec seznamu – neprázdný seznam 4

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



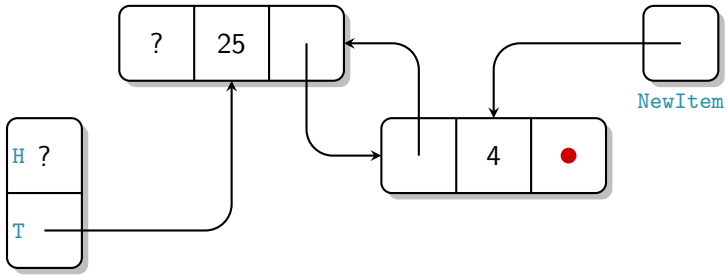
Vložení položky na konec seznamu – neprázdný seznam 5

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



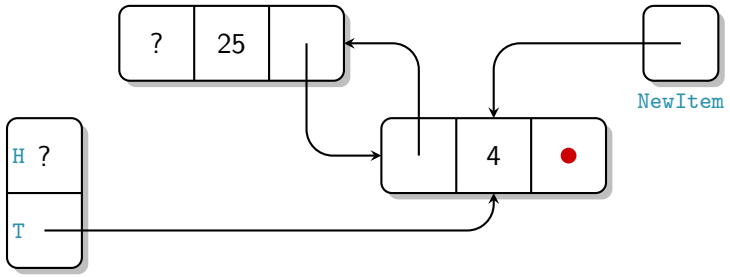
Vložení položky na konec seznamu – neprázdný seznam 6

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



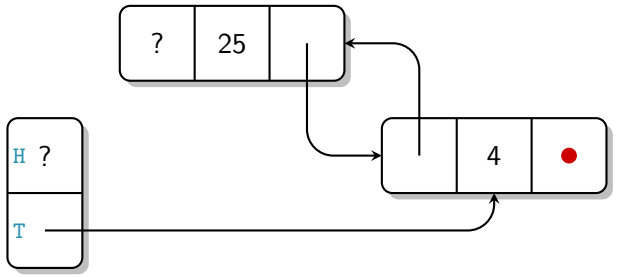
Vložení položky na konec seznamu – neprázdný seznam 7

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



Vložení položky na konec seznamu – neprázdný seznam 8

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



Vložení položky do seznamu – před danou položku

Zde mohou nastat dvě situace:

1. daná položka je hlavou seznamu – využijeme funkci `Prepend` a
2. daná položka je obecně uvnitř seznamu.

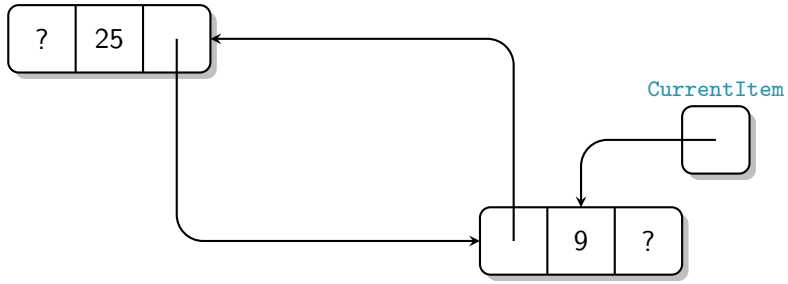
```
64 void InsertBefore(List& L, ListItem* CurrentItem, const string&
       NewValue)
65 {
66    if (CurrentItem != nullptr)
67    {
68       if (CurrentItem == L.Head)
69       {
70          Prepend(L, NewValue);
71       }
72       else
73       {
74          ListItem* NewItem = new ListItem;
75          NewItem->Value = NewValue;
76          ListItem* P = CurrentItem->Prev;
```


Vložení položky do seznamu – před danou položku (pokrač.)

```
77          P->Next = NewItem;
78          NewItem->Prev = P;
79          CurrentItem->Prev = NewItem;
80          NewItem->Next = CurrentItem;
81       }
82    }
83 }
```

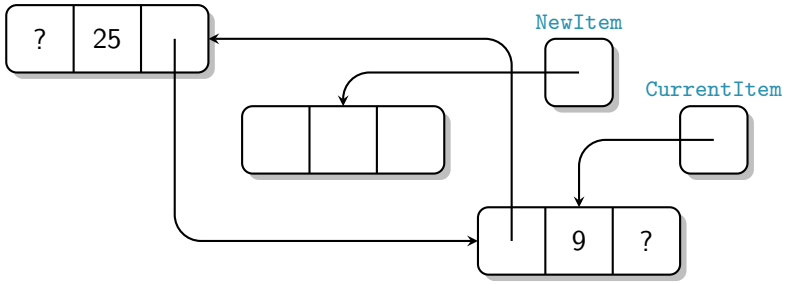

Vložení položky do seznamu před danou položku – obecně 1

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



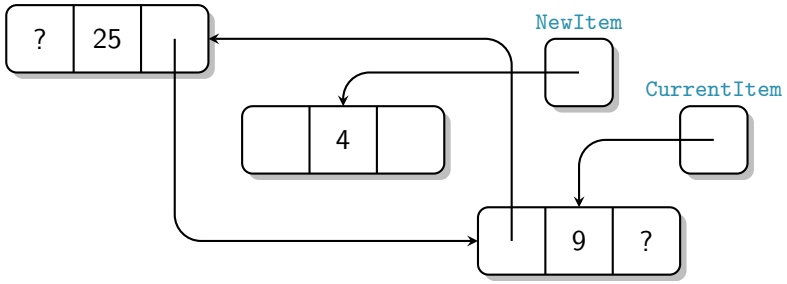
Vložení položky do seznamu před danou položku – obecně 2

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



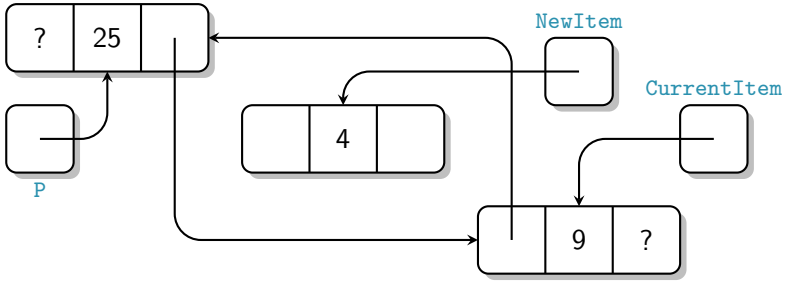
Vložení položky do seznamu před danou položku – obecně 3

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



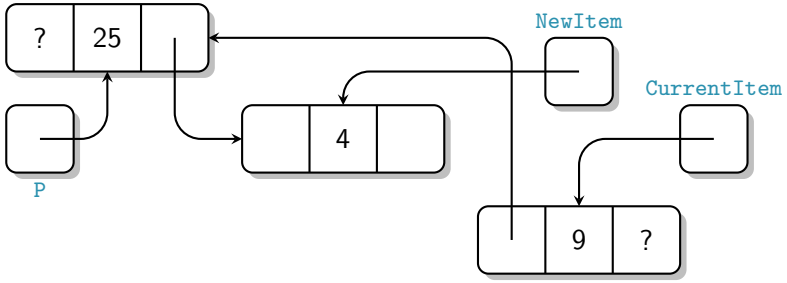
Vložení položky do seznamu před danou položku – obecně 4

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



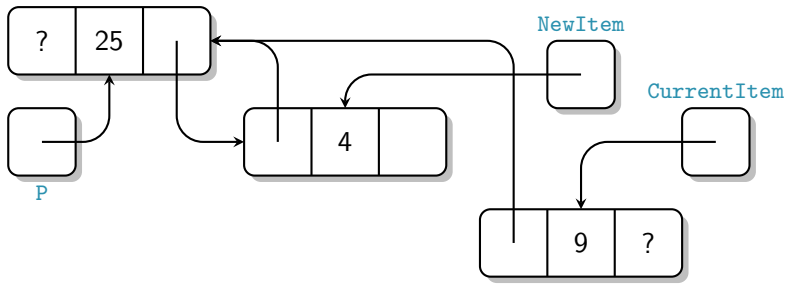
Vložení položky do seznamu před danou položku – obecně 5

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



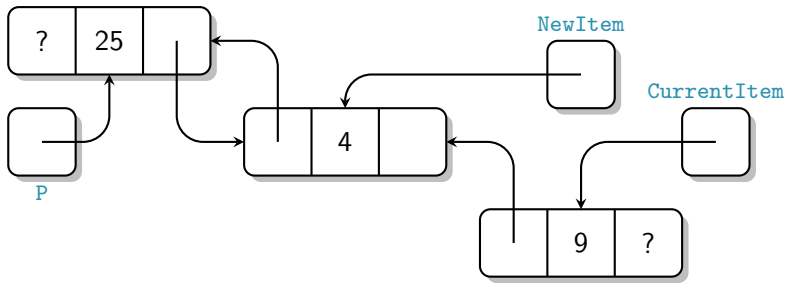
Vložení položky do seznamu před danou položku – obecně 6

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



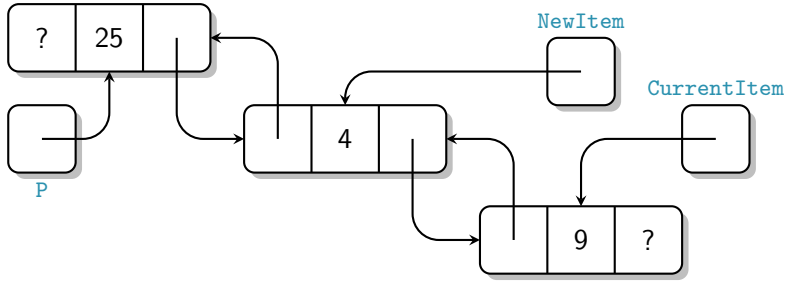
Vložení položky do seznamu před danou položku – obecně 7

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



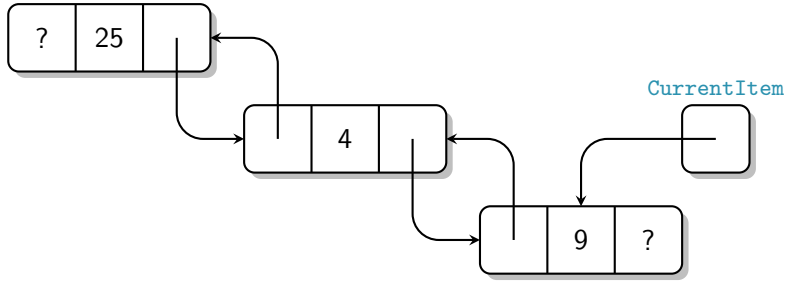
Vložení položky do seznamu před danou položku – obecně 8

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



Vložení položky do seznamu před danou položku – obecně 9

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



Vložení položky do seznamu – za danou položku

Zde mohou nastat dvě situace:

1. daná položka je ocasem seznamu – využijeme funkci `Append` a
2. daná položka je obecně uvnitř seznamu.

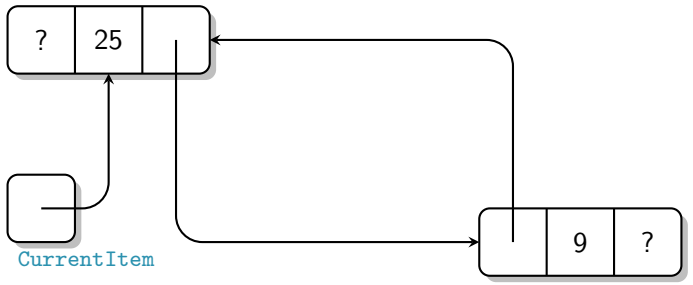
```
85 void InsertAfter(List& L, ListItem* CurrentItem, const string& NewValue
       )
86 {
87    if (CurrentItem != nullptr)
88    {
89       if (CurrentItem == L.Tail)
90       {
91          Append(L, NewValue);
92       }
93       else
94       {
95          ListItem* NewItem = new ListItem;
96          NewItem->Value = NewValue;
97          ListItem* N = CurrentItem->Next;
```


Vložení položky do seznamu – za danou položku (pokrač.)

```
98          N->Prev = NewItem;
99          NewItem->Next = N;
100         CurrentItem->Next = NewItem;
101         NewItem->Prev = CurrentItem;
102      }
103   }
104 }
```

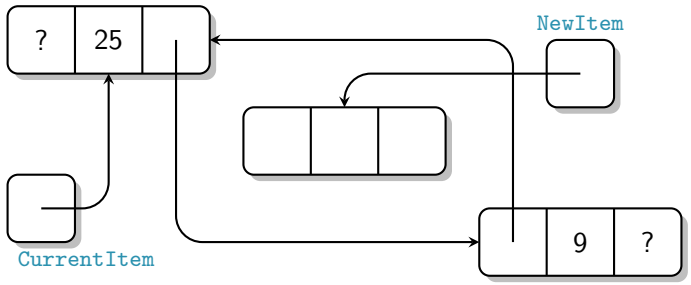

Vložení položky do seznamu za danou položku – obecně 1

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



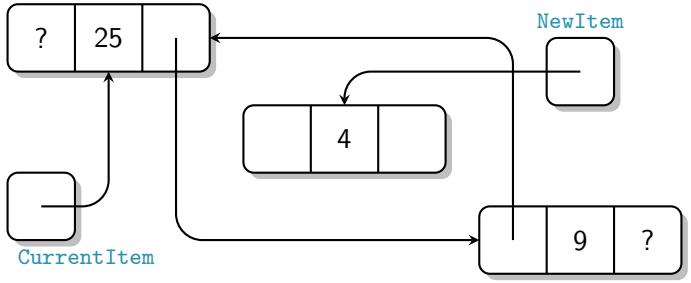
Vložení položky do seznamu za danou položku – obecně 2

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



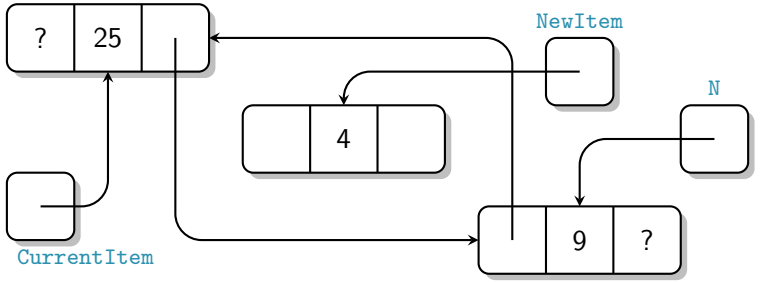
Vložení položky do seznamu za danou položku – obecně 3

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



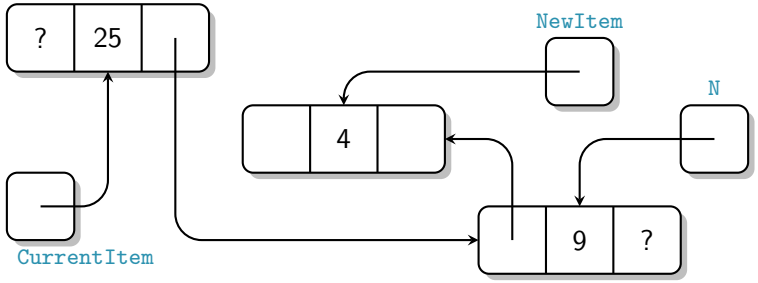
Vložení položky do seznamu za danou položku – obecně 4

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



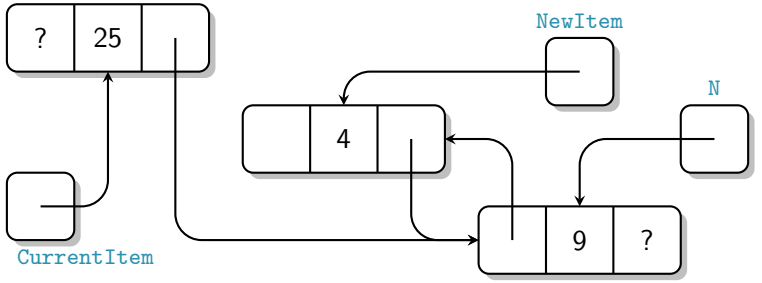
Vložení položky do seznamu za danou položku – obecně 5

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



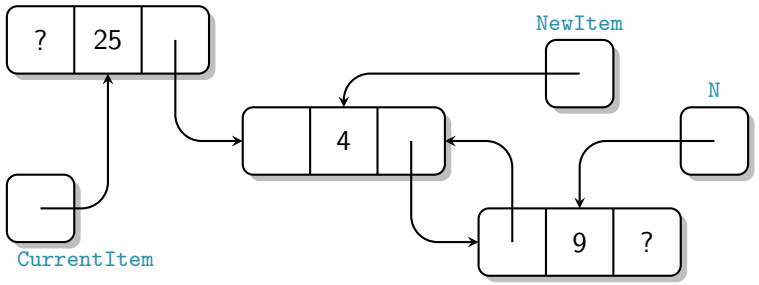
Vložení položky do seznamu za danou položku – obecně 6

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



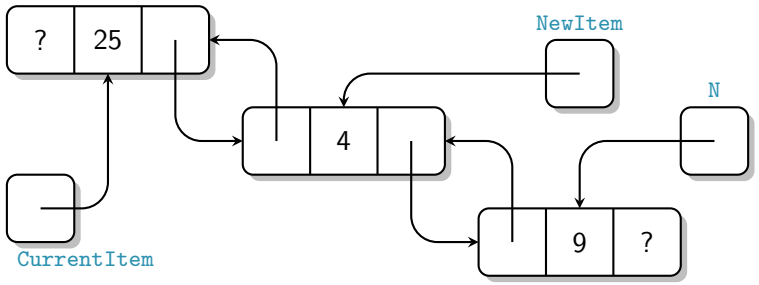
Vložení položky do seznamu za danou položku – obecně 7

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



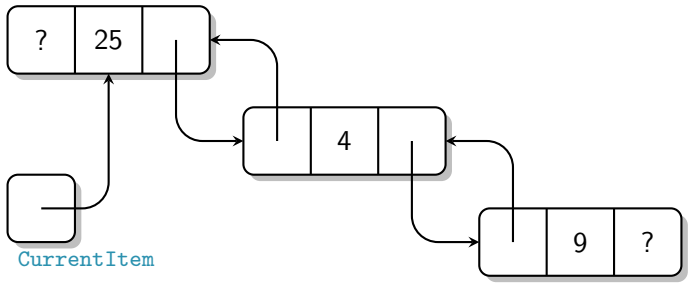
Vložení položky do seznamu za danou položku – obecně 8

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



Vložení položky do seznamu za danou položku – obecně 9

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



Odebrání položky ze seznamu

Odebrání položky ze seznamu můžeme rozdělit na dvě fáze:

1. vlastní odebrání položky ze seznamu – odstranění vazeb mezi ostatními položkami seznamu a odebíranou položkou. Zde mohou nastat celkem čtyři situace:
   - 1.1 seznam obsahuje jedinou položku – řádky 242 až 246,
   - 1.2 odebíraná položka je hlavou seznamu – řádky 249 až 253,
   - 1.3 odebíraná položka je ocasem seznamu – řádky 256 až 260 nebo
   - 1.4 odebíraná položka leží obecně uvnitř seznamu – řádky 263 až 266.
2. uvolnění položky z paměti – řádek 270

Odebrání položky ze seznamu (pokrač.)

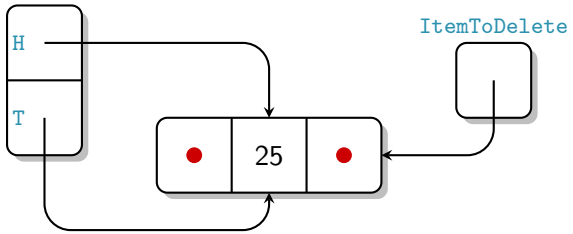
```
240 void InternalRemove(List& L, const ListItem* ItemToDelete)
241 {
242    if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243    {
244       L.Head = nullptr;
245       L.Tail = nullptr;
246    }
247    else
248    {
249       if (ItemToDelete == L.Head)
250       {
251          L.Head = L.Head->Next;
252          L.Head->Prev = nullptr;
253       }
254       else
255       {
256          if (ItemToDelete == L.Tail)
257          {
258             L.Tail = L.Tail->Prev;
```


Odebrání položky ze seznamu (pokrač.)

```
259             L.Tail->Next = nullptr;
260          }
261          else
262          {
263             ListItem* P = ItemToDelete->Prev;
264             ListItem* N = ItemToDelete->Next;
265             P->Next = N;
266             N->Prev = P;
267          }
268       }
269    }
270    delete ItemToDelete;
271 }
```

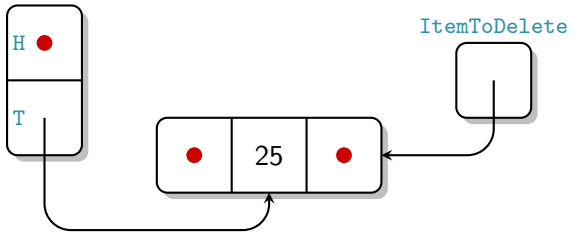

Odebrání položky ze seznamu – jediná položka 1

```
242 if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243 {
244    L.Head = nullptr;
245    L.Tail = nullptr;
246 }
```



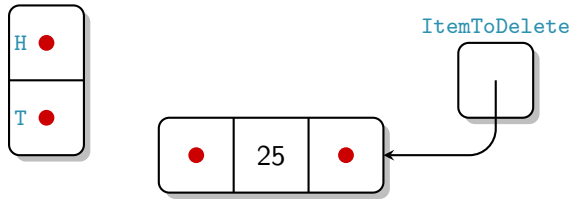
Odebrání položky ze seznamu – jediná položka 2

```
242 if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243 {
244    L.Head = nullptr;
245    L.Tail = nullptr;
246 }
```



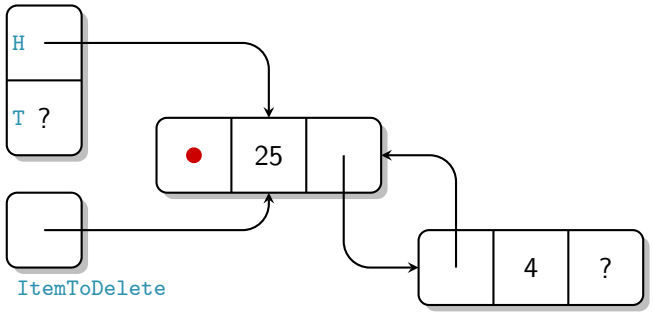
Odebrání položky ze seznamu – jediná položka 3

```
242 if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243 {
244    L.Head = nullptr;
245    L.Tail = nullptr;
246 }
```



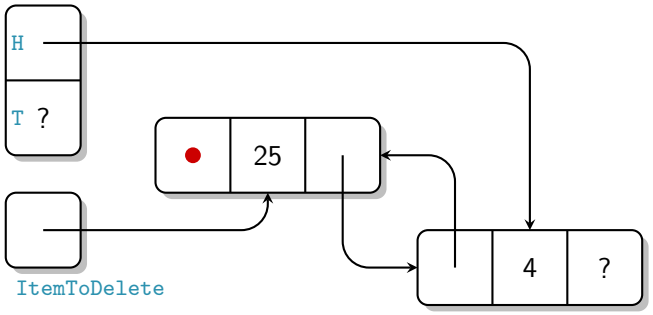
Odebrání položky ze seznamu – hlava 1

```
249 if (ItemToDelete == L.Head)
250 {
251    L.Head = L.Head->Next;
252    L.Head->Prev = nullptr;
253 }
```



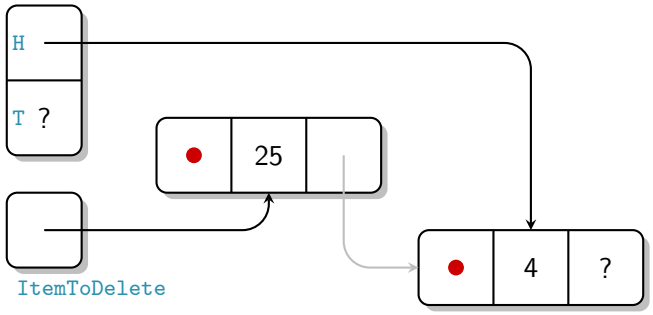
Odebrání položky ze seznamu – hlava 2

```
249 if (ItemToDelete == L.Head)
250 {
251    L.Head = L.Head->Next;
252    L.Head->Prev = nullptr;
253 }
```



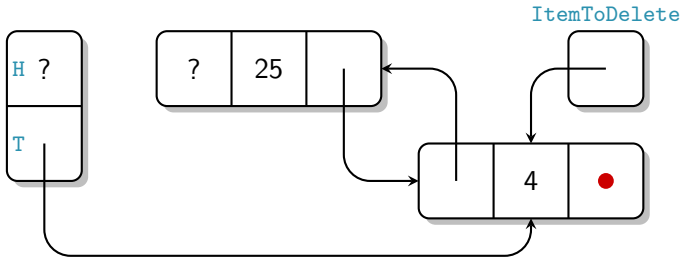
Odebrání položky ze seznamu – hlava 3

```
249 if (ItemToDelete == L.Head)
250 {
251    L.Head = L.Head->Next;
252    L.Head->Prev = nullptr;
253 }
```



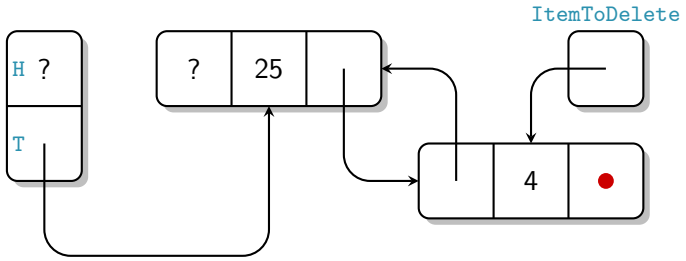
Odebrání položky ze seznamu – ocas 1

```
256 if (ItemToDelete == L.Tail)
257 {
258    L.Tail = L.Tail->Prev;
259    L.Tail->Next = nullptr;
260 }
```



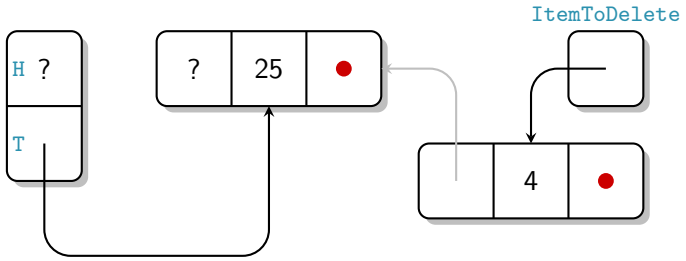
Odebrání položky ze seznamu – ocas 2

```
256 if (ItemToDelete == L.Tail)
257 {
258    L.Tail = L.Tail->Prev;
259    L.Tail->Next = nullptr;
260 }
```



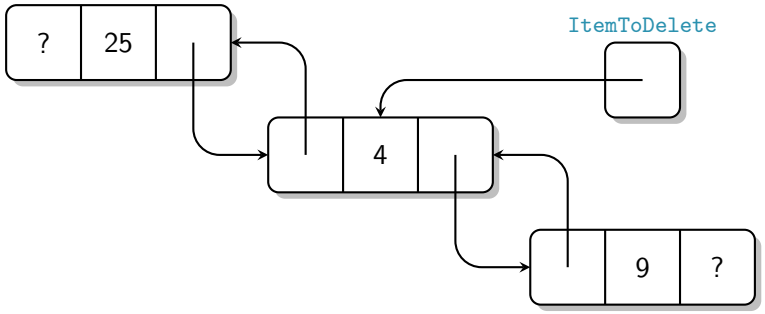
Odebrání položky ze seznamu – ocas 3

```
256 if (ItemToDelete == L.Tail)
257 {
258    L.Tail = L.Tail->Prev;
259    L.Tail->Next = nullptr;
260 }
```



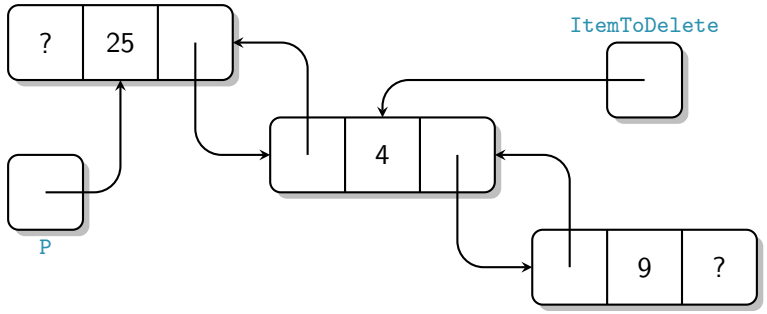
Odebrání položky ze seznamu – obecná pozice 1

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



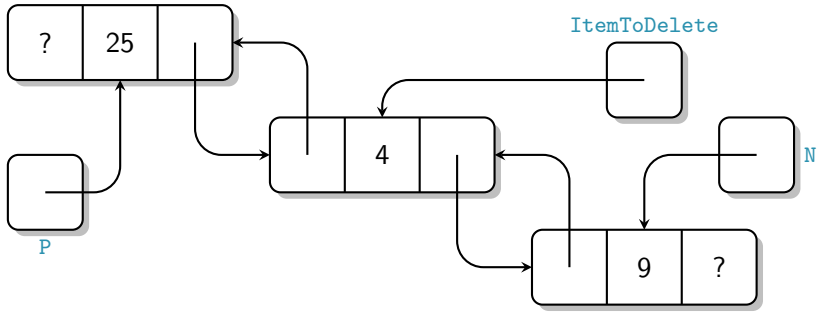
Odebrání položky ze seznamu – obecná pozice 2

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



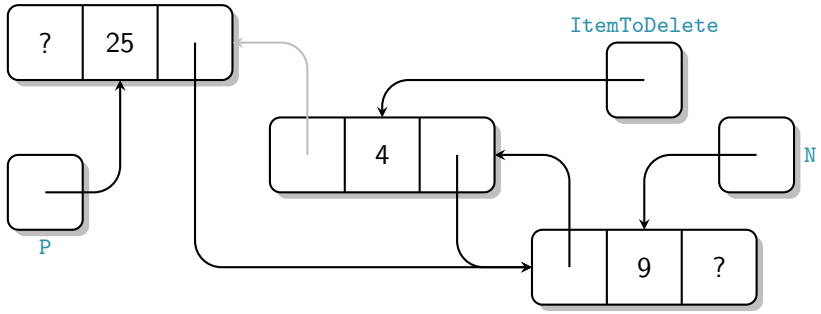
Odebrání položky ze seznamu – obecná pozice 3

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



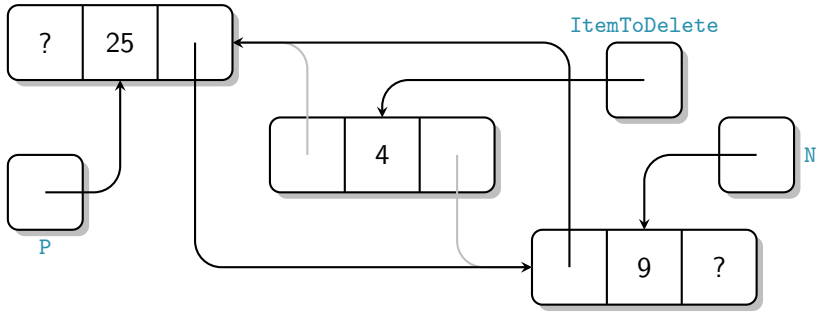
Odebrání položky ze seznamu – obecná pozice 4

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



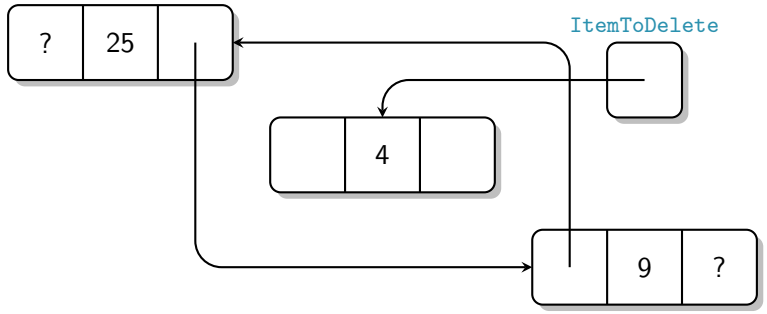
Odebrání položky ze seznamu – obecná pozice 5

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



Odebrání položky ze seznamu – obecná pozice 6

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



Spojová implementace obousměrného seznamu – deklarace

```
1 void Init(List& L);
2 void Init(List& L, const string& InitValue);
3 void Init(List& L, const string InitValues[], const int N);
4 void Init(List& L, const List& Other);
5
6 void Prepend(List& L, const string& NewValue);
7 void Append(List& L, const string& NewValue);
8 void InsertBefore(List& L, ListItem* CurrentItem, const string&
      NewValue);
9 void InsertAfter(List& L, ListItem* CurrentItem, const string& NewValue
      );
10
11 void RemoveHead(List& L);
12 void RemoveTail(List& L);
13 void RemoveFirst(List& L, const string& ValueToRemove);
14 void RemoveLast(List& L, const string& ValueToRemove);
15 void RemoveAll(List& L, const string& ValueToRemove);
16 void Remove(List& L, const ListItem* ItemToRemove);
```


Spojová implementace obousměrného seznamu – deklarace (pokrač.)

```
17 void Clear(List& L);
18
19 ListItem* Search(const List& L, const string& Value);
20 ListItem* ReverseSearch(const List& L, const string& Value);
21
22 bool Contains(const List& L, const string& Value);
23
24 bool IsEmpty(const List& L);
25 int Count(const List& L);
26
27 void Report(const List& L);
28 void ReportStructure(const List& L);
29
30 void InternalCreateSingleElementList(List& L, const string& NewValue);
31 void InternalRemove(List& L, const ListItem* ItemToDelete);
```


Spojová implementace obousměrného seznamu – implementace

```
1 void Init(List& L)
2 {
3    L.Head = nullptr;
4    L.Tail = nullptr;
5 }
6
7 void Init(List& L, const string& InitValue)
8 {
9    InternalCreateSingleElementList(L, InitValue);
10 }
11
12 void Init(List& L, const string InitValues[], const int N)
13 {
14    Init(L);
15    for (int i = 0; i < N; i++)
16    {
17       Append(L, InitValues[i]);
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
18    }
19 }
20
21 void Init(List& L, const List& Other)
22 {
23    Init(L);
24    for (ListItem* p = Other.Head; p != nullptr; p = p->Next)
25    {
26       Append(L, p->Value);
27    }
28 }
29
30 void Prepend(List& L, const string& NewValue)
31 {
32    if (IsEmpty(L))
33    {
34       InternalCreateSingleElementList(L, NewValue);
35    }
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
36    else
37    {
38       ListItem* NewItem = new ListItem;
39       NewItem->Value = NewValue;
40       NewItem->Prev = nullptr;
41       NewItem->Next = L.Head;
42       L.Head->Prev = NewItem;
43       L.Head = NewItem;
44    }
45 }
46
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
54    {
55       ListItem* NewItem = new ListItem;
56       NewItem->Value = NewValue;
57       NewItem->Next = nullptr;
58       L.Tail->Next = NewItem;
59       NewItem->Prev = L.Tail;
60       L.Tail = NewItem;
61    }
62 }
63
64 void InsertBefore(List& L, ListItem* CurrentItem, const string&
       NewValue)
65 {
66    if (CurrentItem != nullptr)
67    {
68       if (CurrentItem == L.Head)
69       {
70          Prepend(L, NewValue);
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
71       }
72       else
73       {
74          ListItem* NewItem = new ListItem;
75          NewItem->Value = NewValue;
76          ListItem* P = CurrentItem->Prev;
77          P->Next = NewItem;
78          NewItem->Prev = P;
79          CurrentItem->Prev = NewItem;
80          NewItem->Next = CurrentItem;
81       }
82    }
83 }
84
85 void InsertAfter(List& L, ListItem* CurrentItem, const string& NewValue
       )
86 {
87    if (CurrentItem != nullptr)
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
88    {
89       if (CurrentItem == L.Tail)
90       {
91          Append(L, NewValue);
92       }
93       else
94       {
95          ListItem* NewItem = new ListItem;
96          NewItem->Value = NewValue;
97          ListItem* N = CurrentItem->Next;
98          N->Prev = NewItem;
99          NewItem->Next = N;
100         CurrentItem->Next = NewItem;
101         NewItem->Prev = CurrentItem;
102      }
103   }
104 }
105
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
106 void RemoveHead(List& L)
107 {
108    if (!IsEmpty(L))
109    {
110       InternalRemove(L, L.Head);
111    }
112 }
113
114 void RemoveTail(List& L)
115 {
116    if (!IsEmpty(L))
117    {
118       InternalRemove(L, L.Tail);
119    }
120 }
121
122 void RemoveFirst(List& L, const string& ValueToRemove)
123 {
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
124    ListItem* p = Search(L, ValueToRemove);
125    if (p != nullptr)
126    {
127       InternalRemove(L, p);
128    }
129 }
130
131 void RemoveLast(List& L, const string& ValueToRemove)
132 {
133    ListItem* p = ReverseSearch(L, ValueToRemove);
134    if (p != nullptr)
135    {
136       InternalRemove(L, p);
137    }
138 }
139
140 void RemoveAll(List& L, const string& ValueToRemove)
141 {
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
142    ListItem* p;
143    while ((p = Search(L, ValueToRemove)) != nullptr)
144    {
145       InternalRemove(L, p);
146    }
147 }
148
149 void Remove(List& L, const ListItem* ItemToRemove)
150 {
151    if (ItemToRemove != nullptr)
152    {
153       InternalRemove(L, ItemToRemove);
154    }
155 }
156
157 void Clear(List& L)
158 {
159    while (!IsEmpty(L))
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
160    {
161       InternalRemove(L, L.Head);
162    }
163 }
164
165 ListItem* Search(const List& L, const string& Value)
166 {
167    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
168    {
169       if (p->Value == Value)
170       {
171          return p;
172       }
173    }
174    return nullptr;
175 }
176
177 ListItem* ReverseSearch(const List& L, const string& Value)
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
178 {
179    for (ListItem* p = L.Tail; p != nullptr; p = p->Prev)
180    {
181       if (p->Value == Value)
182       {
183          return p;
184       }
185    }
186    return nullptr;
187 }
188
189 bool Contains(const List& L, const string& Value)
190 {
191    return Search(L, Value) != nullptr;
192 }
193
194 bool IsEmpty(const List& L)
195 {
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
196    return L.Head == nullptr;
197 }
198
199 int Count(const List& L)
200 {
201    int counter = 0;
202    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
203    {
204       counter += 1;
205    }
206    return counter;
207 }
208
209 void Report(const List& L)
210 {
211    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
212    {
213       cout << p->Value << "\t";
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
214    }
215    cout << endl;
216 }
217
218 void ReportStructure(const List& L)
219 {
220    cout << "L.Head: " << L.Head << endl;
221    cout << "L.Tail: " << L.Tail << endl;
222    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
223    {
224       cout << "Item address: " << p << endl;
225       cout << "Value: " << p->Value << endl;
226       cout << "Prev: " << p->Prev << endl;
227       cout << "Next: " << p->Next << endl;
228       cout << endl;
229    }
230 }
231
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
232 void InternalCreateSingleElementList(List& L, const string& NewValue)
233 {
234    L.Head = L.Tail = new ListItem;
235    L.Head->Value = NewValue;
236    L.Head->Prev = nullptr;
237    L.Head->Next = nullptr;
238 }
239
240 void InternalRemove(List& L, const ListItem* ItemToDelete)
241 {
242    if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243    {
244       L.Head = nullptr;
245       L.Tail = nullptr;
246    }
247    else
248    {
249       if (ItemToDelete == L.Head)
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
250       {
251          L.Head = L.Head->Next;
252          L.Head->Prev = nullptr;
253       }
254       else
255       {
256          if (ItemToDelete == L.Tail)
257          {
258             L.Tail = L.Tail->Prev;
259             L.Tail->Next = nullptr;
260          }
261          else
262          {
263             ListItem* P = ItemToDelete->Prev;
264             ListItem* N = ItemToDelete->Next;
265             P->Next = N;
266             N->Prev = P;
267          }
```


Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
268       }
269    }
270    delete ItemToDelete;
271 }
```


Srovnání implementací seznamu

Pole

```
236 void InternalCreateSingleElementList(List& L, const string& NewValue)
237 {
238    L.Head = L.Tail = InternalAllocateItem(L);
239    L.Items[L.Head].Value = NewValue;
240    L.Items[L.Head].Prev = Nothing;
241    L.Items[L.Head].Next = Nothing;
242 }
```

Spojové struktury

```
232 void InternalCreateSingleElementList(List& L, const string& NewValue)
233 {
234    L.Head = L.Tail = new ListItem;
235    L.Head->Value = NewValue;
236    L.Head->Prev = nullptr;
237    L.Head->Next = nullptr;
238 }
```

Srovnání implementací seznamu (pokrač.)

Pole

```
169 int Search(const List& L, const string& Value)
170 {
171    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
172    {
173       if (L.Items[p].Value == Value)
174       {
175          return p;
176       }
177    }
178    return Nothing;
179 }
```

Spojové struktury

```
165 ListItem* Search(const List& L, const string& Value)
166 {
167    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
168    {
169       if (p->Value == Value)
170       {
171          return p;
172       }
173    }
174    return nullptr;
175 }
```


Srovnání implementací seznamu (pokrač.)

Pole

```
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
54    {
55       int NewItemIndex = InternalAllocateItem(L);
56       L.Items[NewItemIndex].Value = NewValue;
57       L.Items[NewItemIndex].Prev = L.Tail;
58       L.Items[NewItemIndex].Next = Nothing;
59       L.Items[L.Tail].Next = NewItemIndex;
60       L.Tail = NewItemIndex;
61    }
62 }
```

Spojové struktury

```
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L,
               NewValue);
52    }
53    else
54    {
55       ListItem* NewItem = new ListItem;
56       NewItem->Value = NewValue;
57       NewItem->Next = nullptr;
58       L.Tail->Next = NewItem;
59       NewItem->Prev = L.Tail;
60       L.Tail = NewItem;
61    }
62 }
```


Kontrolní otázky

1. Máme danou strukturu a čtyři proměnné:

   ```
   1 struct MyStruct
   2 {
   3    int Id;
   4    double Value;
   5 };
   6
   7 MyStruct a, b;
   8 MyStruct *p, *q;
   ```

   Co se stane pokud provedu `a = b;`? A co pokud provedu `p = q;`?
2. Jak alokuji dynamicky strukturu z předchozí otázky? Pomocí jakého operátoru?
3. Jak uvolním z paměti dynamicky alokovanou strukturu, kterou jsem vytvořil v předchozí otázce?
4. Mám deklarovanou proměnnou `MyStruct a;`. Jak se dostanu k položce `Value` v proměnné `a`?

Kontrolní otázky (pokrač.)

5. Mám deklarovanou proměnnou `MyStruct *p;`. Jak se dostanu k položce `Value` v proměnné `p` pomocí jediného operátoru (bez dereference)?
6. Mám deklarovanou proměnnou `MyStruct *p;`. Jak se dostanu k položce `Value` v proměnné `p` pomocí operátoru dereference?
7. Mám následující kód:

   ```
   1 double *p = new double;
   2 delete p;
   3 *p = 15;
   ```

   Co tento kód způsobí?
8. Mám následující kód:

   ```
   1 double *p = new double;
   2 p = NULL;
   ```

   Co tento kód způsobí? K čemu v paměti dojde?

Kontrolní otázky (pokrač.)

9. Mám následující kód:

```
1 double *p = new double;
2 p = NULL;
3 *p = 15;
```

   Co tento kód způsobí? K čemu v paměti dojde?

10. Mám seznam implementovaný pomocí pole. Musím tento seznam (především to pole) explicitně uvolňovat z paměti pomocí operátoru `delete`?
11. Mám seznam implementovaný pomocí dynamicky alokovaných struktur. Musím tento seznam explicitně uvolňovat z paměti pomocí operátoru `delete`?
12. Co je to zarážka (angl. sentinel)? Co nahrazuje?
13. K čemu slouží zarážka?

Kontrolní otázky (pokrač.)

14. Jak se změní algoritmus vkládání prvku na začátek obousměrného seznamu, pokud využijeme zarážku?
15. Jak se změní algoritmus smazání prvku z obousměrného seznamu, pokud využijeme zarážku?
16. Kdy je vhodné využít zarážku? Zaměřte se na spotřebovanou paměť a počet prvků v uvažovaném seznamu.
17. Kdy je vhodné použít seznam implementovaný v poli a kdy pomocí dynamických struktur? Zaměřte se na rychlost operací vložení (s tím spojené alokování položky), operací mazání (s tím spojené dealokovaní položky). Dále se zaměřte na možnou změnu kapacity seznamu.
18. Jestliže budu k seznamu přistupovat pouze pomocí funkcí `Insert`, `Delete` nebo `Search` a změním implementaci seznamu. Musím měnit i okolní kód, kde se seznam využívá?
19. Pokud implementuji frontu pomocí dynamických struktur, potřebuji kruhový buffer a podobné „triky“ s polem?

Kontrolní otázky (pokrač.)

20. Zkuste navrhnout efektivní algoritmus pro třídění seznamu. Uvědomte si, že k položkám seznamu můžeme přistupovat pouze sekvenčně, jako například k datům v souboru. Jaký Vám známý algoritmus pro třídění lze na tento problém aplikovat?
21. Zkuste navrhnout rekurzívní algoritmus pro vyhledání prvku v obousměrném seznamu.
22. Jak spočítáte počet položek v seznamu? Má na tento algoritmus vliv zda je seznam obousměrný nebo jednosměrný?
23. Jak byste v obousměrném seznamu implementovali mazání k-té položky.
24. Popište algoritmus mazání položky z obousměrného seznamu. Položka určená ke smazání se musí nejprve vyhledat podle dat uložených v položce.
25. Popište algoritmus vkládání na začátek obousměrného seznamu.

Děkuji za pozornost

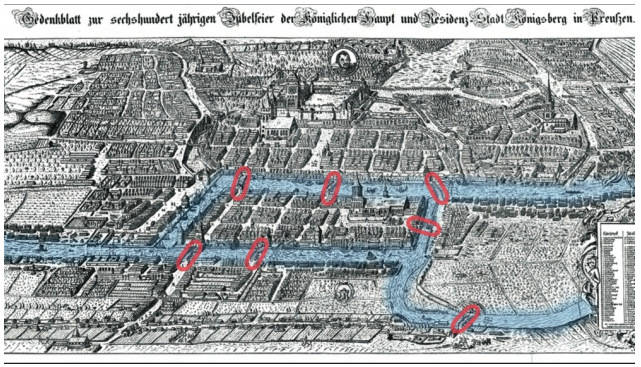
Grafy

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Sedm mostů města Královce

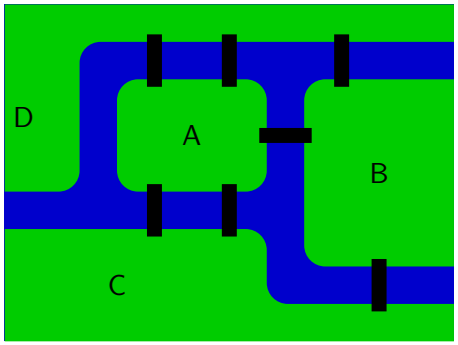
Pruské město Královec (též Königsberg, nyní Kaliningrad na území Ruska) leží na řece Pregole, která vytváří dva ostrovy. Ostrovy byly s ostatním městem spojeny sedmi mosty.



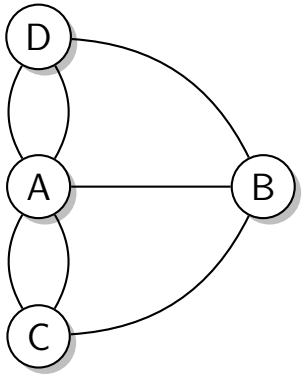
Otázka: Je možné všechny mosty přejít tak, aby ten, kdo se o to pokouší, vstoupil na každý most pouze jednou?

Sedm mostů města Královce

Schematická mapa

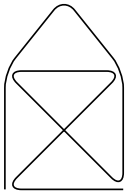


Graf



Sedm mostů města Královce – rozuzlení

- Řešení úlohy publikoval Leonhard Euler v roce 1736.
- Zadání formuloval pomocí teorie grafů, kterou při této příležitosti vytvořil.
- Nezáleží na fyzických tvarech pevniny nebo mostů, že záleží jen propojení pevnin (vrcholy) mosty (hrany).
- Takto vytvořený graf **nelze projít** pomocí eulerovského tahu – to jest tahu, který obsahuje každou hranu právě jednou.
- Existuje-li v grafu uzavřený eulerovský tah, nazýváme tento graf rovněž eulerovský. Eulerovské grafy lze nakreslit „jedním tahem“.



Teorie grafů – další historické úlohy

V roce 1845 publikoval **Gustav Kirchhoff** zákony, které platí v elektrických obvodech a slouží k výpočtu napětí a proudu v jednotlivých větvích obvodu.

V roce 1852 předložil Francis Guthrie takzvaný **problém čtyř barev** – tedy otázku, zda je možné obarvit libovolnou politickou mapu pomocí nejvýše čtyř barev tak, aby každé dvě sousední země (které mají společnou hranici delší než jediný bod) měly odlišnou barvu. Byl vyřešen až o více než sto let později, přičemž pro jeho řešení bylo zavedeno mnoho zásadních konceptů teorie grafů.

Grafy – motivace

- Jedná se o reprezentaci množiny objektů, u které chceme znázornit, že některé prvky jsou propojeny.
- Objektům se přiřadí vrcholy a jejich propojení značí hrany mezi nimi.
- Grafy slouží jako abstrakce mnoha různých problémů.
- Často se jedná o zjednodušený model nějaké skutečné sítě (například dopravní), který zdůrazňuje topologické vlastnosti objektů (vrcholů) a zanedbává geometrické vlastnosti, například přesnou polohu.

Grafy – motivace

Příklady abstrakce

1. kousek železniční sítě ČD, graf jako mapa, reálné názvy stanic, zachován fyzický tvar tratí,
2. zachováme topografické rozložení stanic a jejich názvy, tratě překreslíme úsečkami,
3. zachováme pouze názvy stanic, jejich rozložení můžeme libovolně měnit, u některých nakreslení grafu se budou hrany křížit u jiných ne, některá nakreslení grafu jsou pro lidské oko přehlednější
4. vynecháme názvy stanic a použijeme místo nich přirozená čísla.

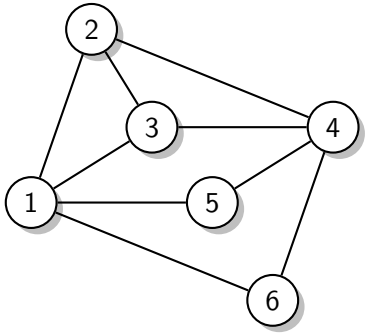
Další možné abstrakce

Vcholy grafu nemusí být nutně stanice. Graf můžeme chápat i tak, že vrcholy jsou jednotlivé tratě (čísla tratí). Dvě tratě spojíme hranou, pokud tyto dvě tratě na sebe navazují. V tomto případě hrany v grafu vyjadřují logický vztah, binární relaci „být propojen“.

Neorientovaný graf

Definice

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



Příklad

$G = (V, E)$

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

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

Neorientovaný graf

Poznámky

- Budeme uvažovat pouze **konečné grafy**, tedy grafy s konečnou množinou vrcholů V.
- Nekonečné grafy necháme stranou.
- Vzhledem k tomu, že jiné než neorientované grafy nebudeme definovat, můžeme označení neorientovaný vynechávat.

Literatura

Jiří Demel: Grafy a jejich aplikace, Academia, Praha, 2002

Neorientovaný graf

Hrany grafu

- Mějme hranu $e \in E$, kde $e = \{u, v\}$. Vrcholům u a v říkáme **krajní vrcholy** hrany e. Říkáme také, že jsou **incidentní** (nebo že **incidují**) s hranou e.
- O hraně e pak říkáme, že je **incidentní** s těmito vrcholy nebo také že **spojuje** tyto vrcholy.

Definice

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

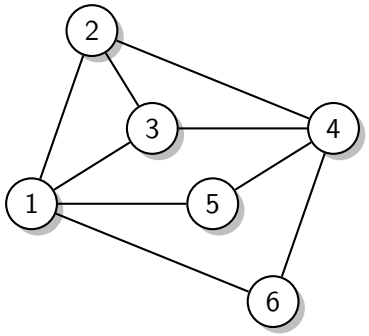
Stupeň vrcholu

Definice

Stupněm vrcholu 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

Stupeň vrcholu (pokrač.)

Věta

Součet stupňů vrcholů libovolného 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). □

Podgraf

Definice

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

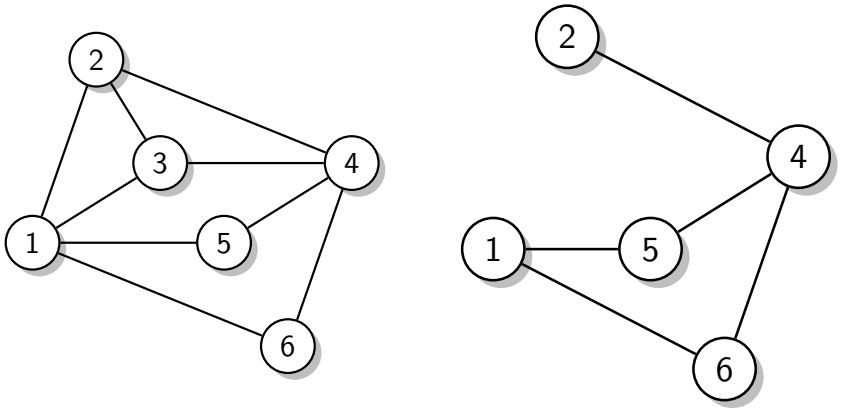
1. $V_H \subseteq V_G$
2. $E_H \subseteq E_G$
3. Hrany grafu H mají oba vrcholy v H.

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

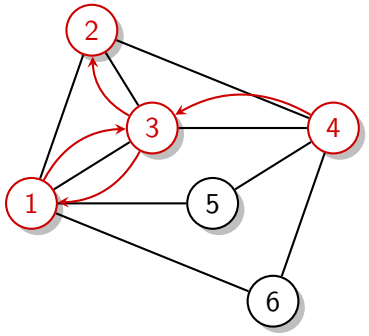


Sled

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**.

Příklad



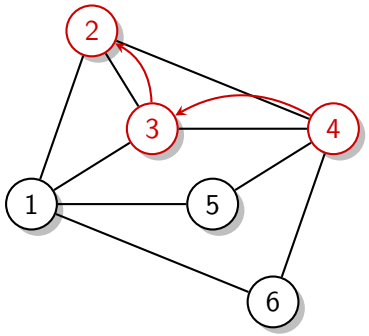
Sled
4 $\{4,3\}$ 3 $\{3,1\}$ 1 $\{1,3\}$ 3 $\{3,2\}$ 2

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**.

Příklad



Cesta
4 3 2

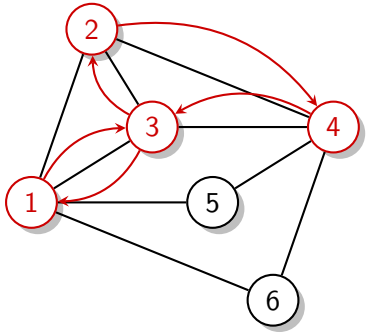
Z faktu, že se v cestě neopakují vrcholy, vyplývá, že se v ní neopakují ani hrany. Každá cesta je tedy zároveň i sledem.

Uzavřený sled

Definice

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

Příklad



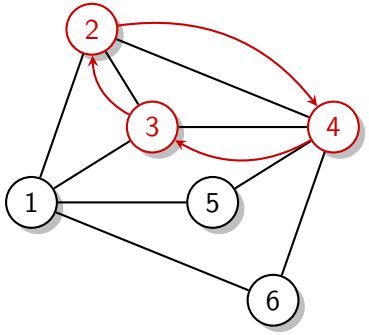
Uzavřený sled
4 $\{4,3\}$ 3 $\{3,1\}$ 1 $\{1,3\}$ 3 $\{3,2\}$ 2 $\{2,4\}$ 4

Kružnice

Definice

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

Příklad



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.

Pojmy založené na cestách

Definice

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

Definice

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

Věta

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

Důkaz.

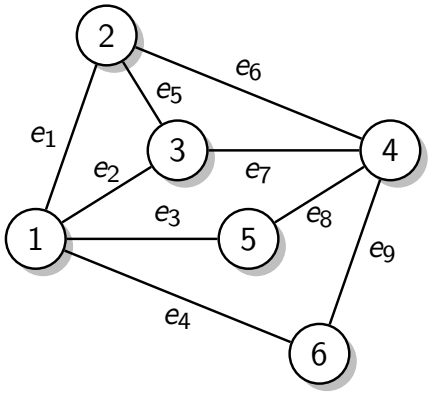
Zřejmý. □

Způsoby zadání 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 slopuců odpovídá počtu hran.
- Pokud je vrchol incidentní s hranou, je na dané pozici jednička, jinak nula.

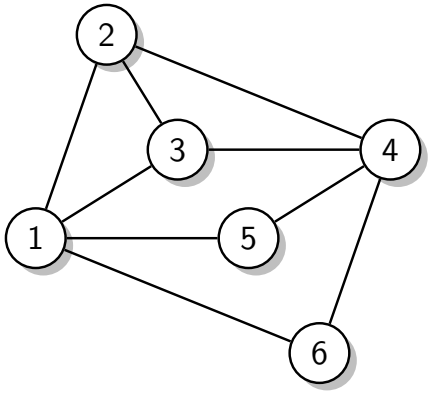


Matice incidence

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

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.

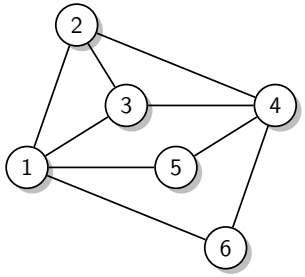


Matice sousednosti

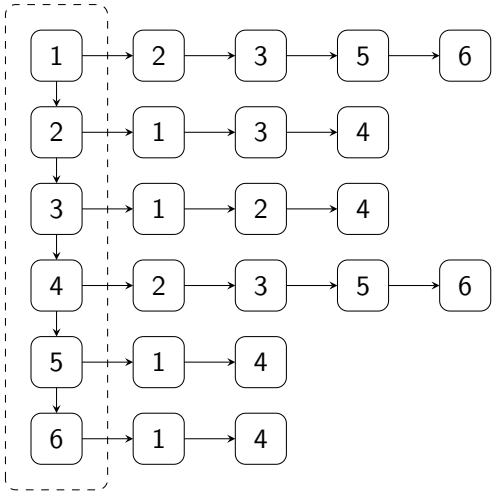
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Seznam sousedících vrcholů

Graf



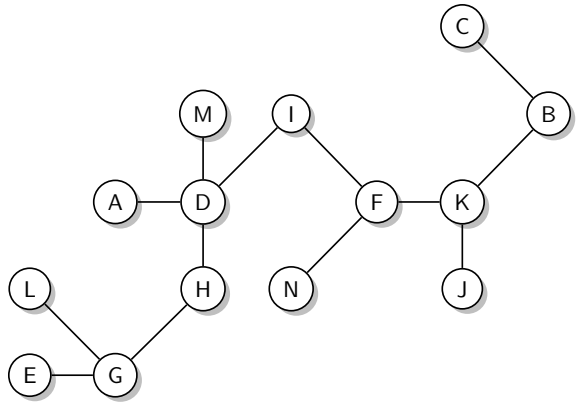
Seznam sousedících vrcholů



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 – 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.

Vlastnosti volného stromu

Věta

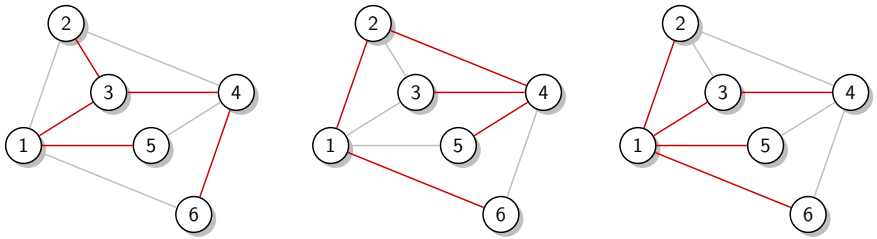
Nechť $G = (V, E)$ je neorientovaný graf, potom následující tvrzení jsou ekvivalentní:

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.



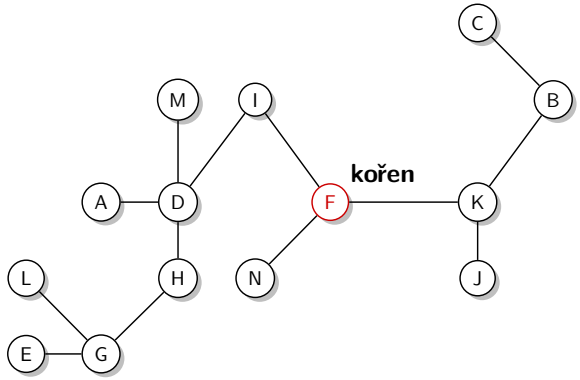
Poznámky

- 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.

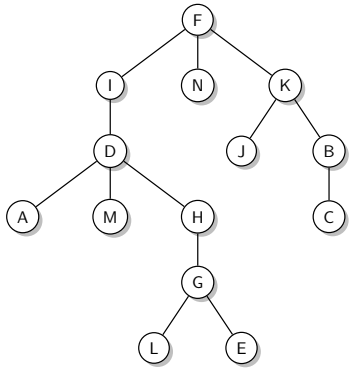


Poznámka

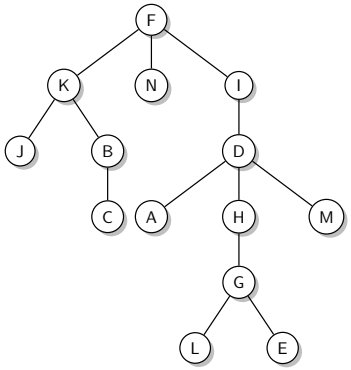
Někdy se používá také označení **zakořeněný strom**.

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.

Předchůdce Libovolný vrchol y na jednoznačné cestě od kořene r do vrcholu x se nazývá **předchůdce** vrcholu x.

Následovník Jestliže y je předchůdce x, potom x se nazývá **následovník** vrcholu y.

Rodič, potomek 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.

Sourozenci Dva vrcholy mající stejného rodiče se nazývají **sourozenci**.

List Vrchol bez potomků se nazývá vnější vrchol nebo-li **list**.

Vnitřní vrchol 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

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

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

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

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“.

Kontrolní otázky

1. Definujte pojem neorientovaný graf.
2. Čím se liší orientovaný graf od neorientovaného?
3. Co je to stupeň uzlu?
4. Co je to sled?
5. Co je to cesta?
6. Co je to délka cesty?
7. Co je to uzavřený sled, uzavřená cesta? Jaké je alternativní pojmenování uzavřené cesty?
8. Co znamená, že graf je acyklický?
9. Co znamená, že graf je souvislý?
10. Jak zní definice volného stromu?
11. Kolik existuje mezi dvěma vrcholy volného stromu cest?

Kontrolní otázky (pokrač.)

12. Kolik hran musíme minimálně přidat do volného stromu, abychom dostali graf s kružnicí?
13. Kolik hran musíme z volného stromu nejméně odebrat, abychom dostali nesouvislý graf?
14. Má-li volný strom $|V|$ vrcholů, kolik má hran?
15. Co je to kořenový strom?
16. Vysvětlete pojem předchůdce uzlu.
17. Vysvětlete pojem následovník uzlu.
18. Vysvětlete pojem rodič uzlu.
19. Vysvětlete pojem potomek uzlu.
20. Vysvětlete pojem sourozenec uzlu.
21. Vysvětlete pojem list.
22. Vysvětlete pojem vnitřní uzel.

Kontrolní otázky (pokrač.)

23. Jak se liší definice stupně uzlu pro obecný graf a kořenový strom?
24. Co je to hloubka uzlu.
25. Jak je definována výška stromu?
26. Co je to seřazený strom?
27. Lze v seřazeném stromu u uzlu s jediným potomkem rozlišit, jestli je levý nebo pravý, první či druhý?

Děkuji za pozornost

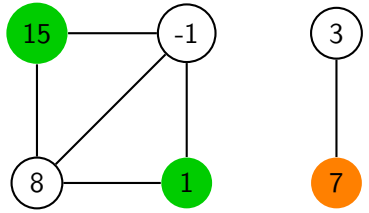
Algoritmy průchodů grafem

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Motivační úlohy



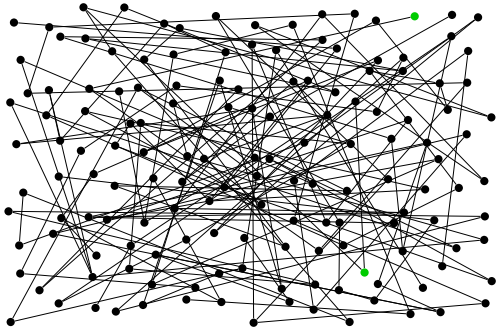
Možné otázky

1. Existuje cesta mezi zelenými vrcholy?
2. Existuje cesta mezi některým zeleným vrcholem a oranžovým vrcholem?
3. Kolik je v grafu zelených vrcholů?
4. Jaká je průměrná hodnota ve vrcholu?

Řešení!? Řešení je přece ihned vidět!

- Co však s grafem o enormním počtu vrcholů? Tisíce, miliony vrcholů.
- Graf má sice jen několik vrcholů, ale potřebuji jich zpracovat několik desítek tisíc. Vše prohlížet ručně!?

Motivační úlohy – aneb co s tímto grafem!?



Dokážete to?

1. Kolik má graf vrcholů?
2. Kolik má hran?
3. Existuje cesta mezi zelenými vrcholy?
4. Existuje v grafu cyklus?
5. Je tento graf souvislý?

Hledané algoritmy

- systematické navštívení všech vrcholů grafu – iterátor a jako důsledek
- vyhledání cesty mezi dvěma zadanými vrcholy či
- nalezení souvislých komponent grafu.

Prohledávání grafu

- Základním účelem prohledávání grafů je zjišťování dostupnosti – do kterých vrcholů se lze ze zadaného počátečního vrcholu dostat.
- Dalším cílem je vykonání nějaké akce s každým vrcholem grafu.
- Z hlediska datových struktur lze prohledávání grafu chápat jako iterátor na grafu.

Prohledávání grafu

- Je-li graf přehledně nakreslen a není příliš rozsáhlý, lze tyto úlohy vyřešit „pouhým pohledem“.
- Problém nastává s rozsáhlými grafy.
- A další problém nastává v případě počítačového zpracování – je nutné vyvinout algoritmus.
- Uvedeme si tři způsoby prohledávání:
  1. značkování vrcholů,
  2. prohledávání do šířky a
  3. prohledávání do hloubky.

Značkování vrcholů

- Vrcholům přiřazujeme značky.
- Má-li vrchol značku, znamená to, že do něj vede nějaká cesta z daného výchozího vrcholu s.
- Algoritmus skončí po $|V| - 1$ opakováních kroků 2 a 3.
- Po ukončení algoritmu budou označkovány všechny vrcholy do nichž vede cesta z vrcholu s.

Algoritmus značkování vrcholů

1. **Inicializace**: Označkujeme vrchol s, ostatní vrcholy jsou beze značek.
2. **Výběr hrany**: Vybereme libovolnou hranu e, jejíž jeden vrchol má značku a druhý vrchol nikoli. Pokračujeme podle kroku 3. Jestliže taková hrana neexistuje, algoritmus končí.
3. **Značkování**: Označkujeme dosud neoznačkovaný vrchol hrany e a pokračujeme v algoritmu podle kroku 2.

Značkování vrcholů - praktická realizace

- Nejjednodušší je opakovaně procházet seznam hran grafu, pro každou hranu testovat, zda ji lze použít pro označkování dalšího vrcholu.
- Nevýhoda spočívá v opakování průchodu seznamem hran.
- Lze vylepšit seznamem vrcholů, které čekají na zpracování – zpracování spočívá v označkování všech vrcholů kam se lze dostat hranami ze zpracovávaného vrcholu.
- Odpadá nutnost procházet celý seznam hran.
- Prohledávání do šířky a do hloubky v podstatě závisí na tom, jak budeme tento pomocný seznam organizovat.

Prohledávání grafu do šířky

- anglicky Breadth-first Search (BFS),
- asi nejjednodušší algoritmus na prohledávání grafu,
- je základem pro mnoho dalších algoritmů – Jarníkův (Primův) výpočet minimální kostry grafu, Dijkstrův algoritmus nalezení nejkratší cesty,

Idea algoritmu

1. nejprve označkujeme vrchol s,
2. pak označkujeme všechny jeho sousedy,
3. pak sousedy sousedů a tak dále ...

Algoritmus tedy můžeme představit tak, že se grafem rozlévá pomyslná tekutina, která postupně, v jakýchsi vlnách, zaplavuje dostupnou část grafu.

Prohledávání grafu do šířky

- Každému vrcholu přiřadíme jeden ze tří možných stavů:
  - unknown – vrchol ještě nebyl navštíven, algoritmus k němu ještě nedošel.
  - discovered – vrchol byl již objeven, čeká ve frontě na zpracování.
  - finished – zpracování vrcholu bylo dokončeno.
- Každý vrchol je počátku ve stavu unknown, pak přejde do stavu discovered a nakonec se ocitne ve stavu finished.
- Vrcholy ve stavu discovered tvoří hranici mezi vrcholy ve stavu unknown a finished. Tato hranice se neustále rozšiřuje.

Stav vrchol u	Možné stavy jeho sousedů		
unknown	unknown	discovered	
discovered	unknown	discovered	finished
finished		discovered	finished

Prohledávání grafu do šířky

- Graf $G = (V, E)$
- Počáteční vrchol s, $s \in V$.
- Pomocné datové struktury:
  - *state* – tabulka se stavy všech vrcholů,
  - π – o každý vrchol u je v tabulce π uložen jeho předchůdce, pokud neexistuje obsahuje tabulka hodnotu nothing,
  - d – tabulka obsahuje pro každý vrchol jeho vzdálenost od vrcholu s,
  - Q – fronta vrcholů ve stavu discovered.

Algoritmus průchodu grafem do šířky

```
   Input  : Graf G(V, E), počáteční vrchol s ∈ V
   Output: BF-strom
20 foreach u ∈ V/{s} do
21 |   state [u] ← unknown;
22 |   d [u] ← ∞;
23 |   π [u] ← nothing;
24 end
25 Q ← ∅;
26 state [s] ← discovered;
27 d [s] ← 0;
28 π [s] ← nothing;
29 Enqueue(Q, s);
```


Algoritmus průchodu grafem do šířky (pokrač.)

```
30 while Q ≠ ∅ do
31 |   u ← Dequeue(Q);
32 |   foreach v ∈ Adj(G, u) do
33 |   |   if state [v] = unknown then
34 |   |   |   state [v] ← discovered;
35 |   |   |   d [v] ← d [u] + 1;
36 |   |   |   π [v] ← u;
37 |   |   |   Enqueue (Q,v);
38 |   |   end
39 |   end
40 |   state [u] ← finished;
41 end
```


Animace průchodu grafem do šířky – legenda

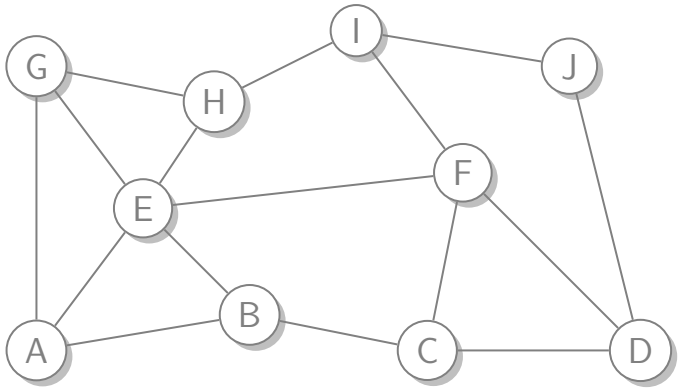
Vrcholy grafu:

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

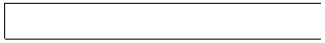
Hrany grafu:

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

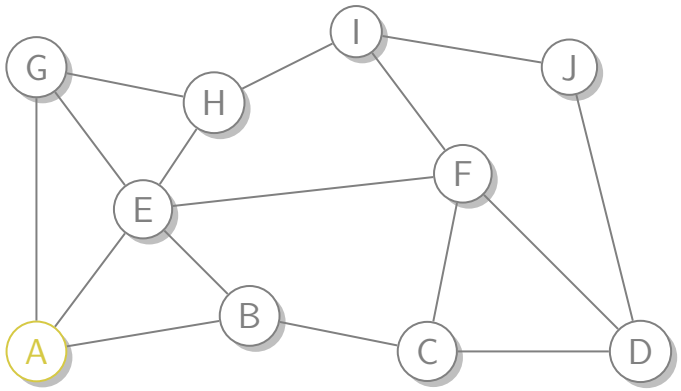
Průchod grafem do šířky, krok 1



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

Q 

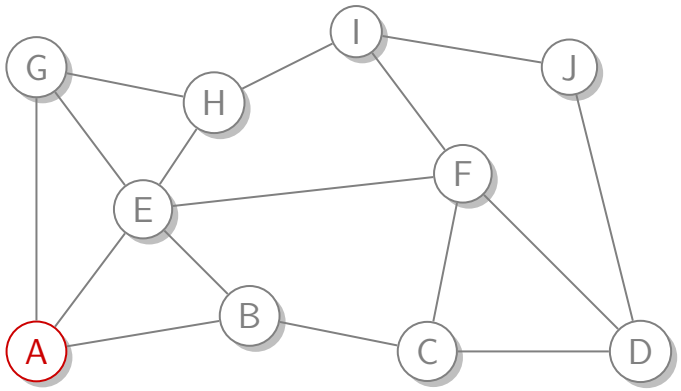
Průchod grafem do šířky, krok 2



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

Q | A |

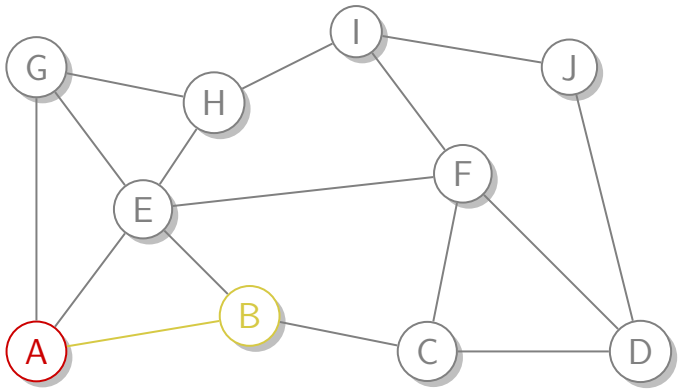
Průchod grafem do šířky, krok 3



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

Q |

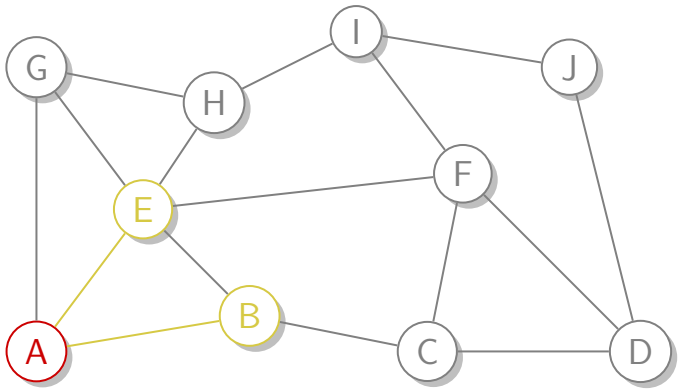
Průchod grafem do šířky, krok 4



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

Q | B |

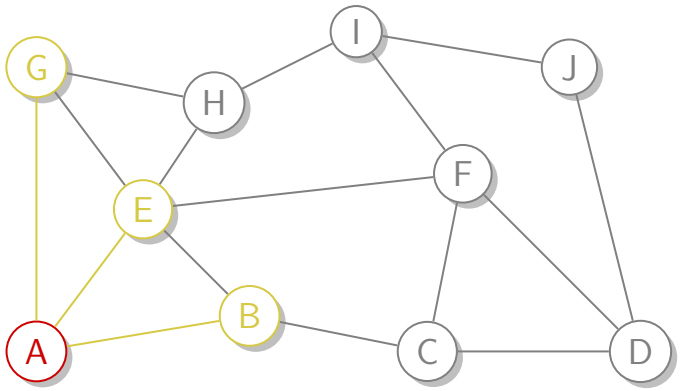
Průchod grafem do šířky, krok 5



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

Q | B | E |

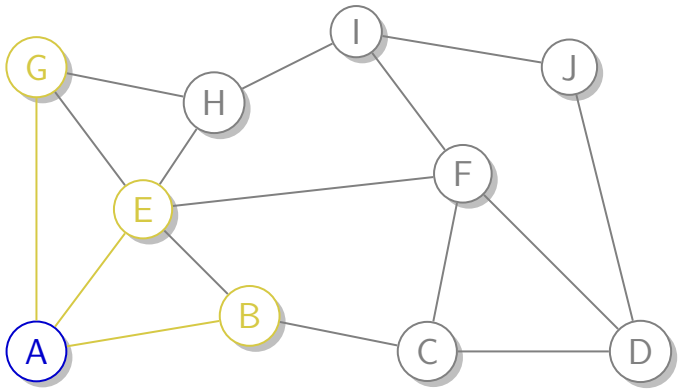
Průchod grafem do šířky, krok 6



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

Q | B | E | G |

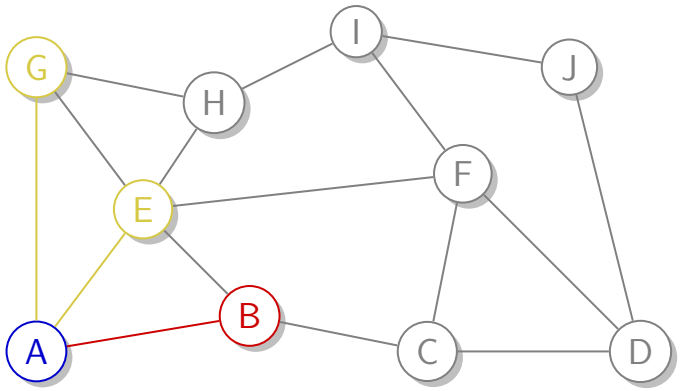
Průchod grafem do šířky, krok 7



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

Q | B | E | G |

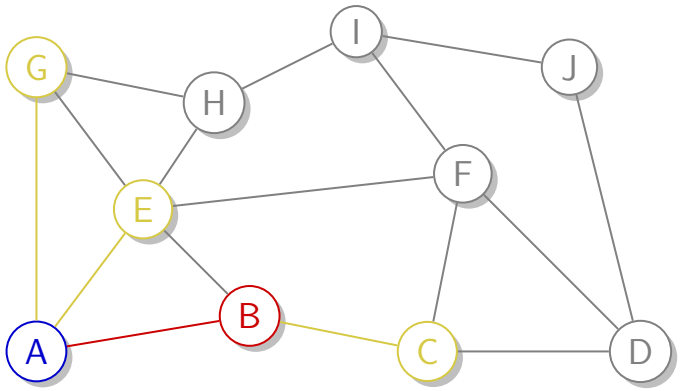
Průchod grafem do šířky, krok 8



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

Q | E | G |

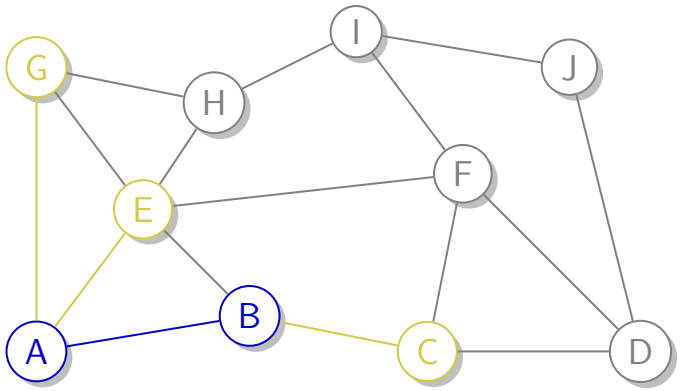
Průchod grafem do šířky, krok 9



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

Q | E | G | C |

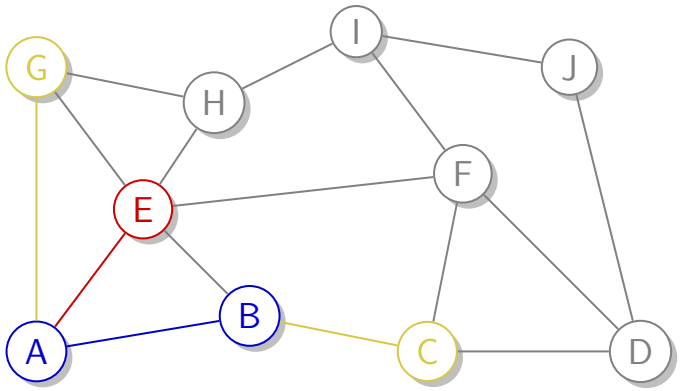
Průchod grafem do šířky, krok 10



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

Q | E | G | C |

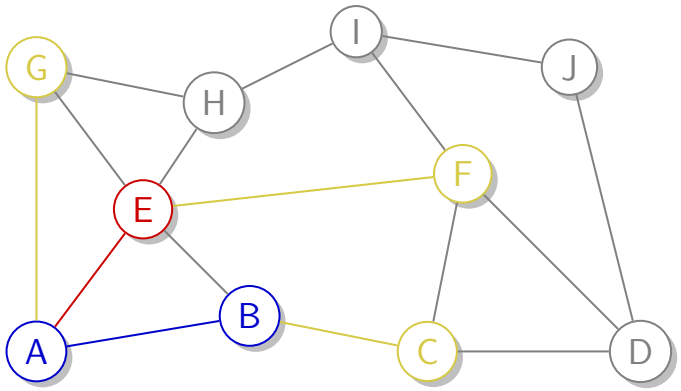
Průchod grafem do šířky, krok 11



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

Q | G | C |

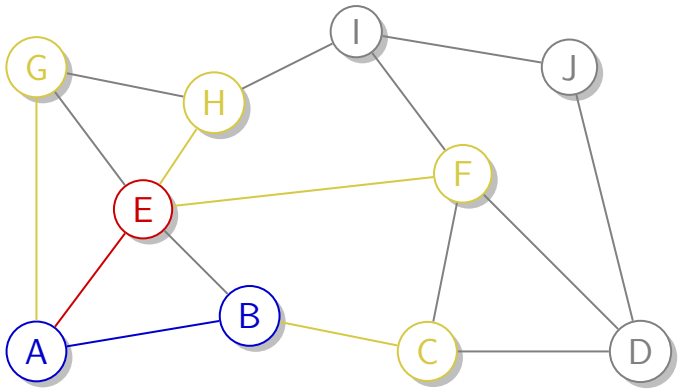
Průchod grafem do šířky, krok 12



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	∞	/

Q | G | C | F |

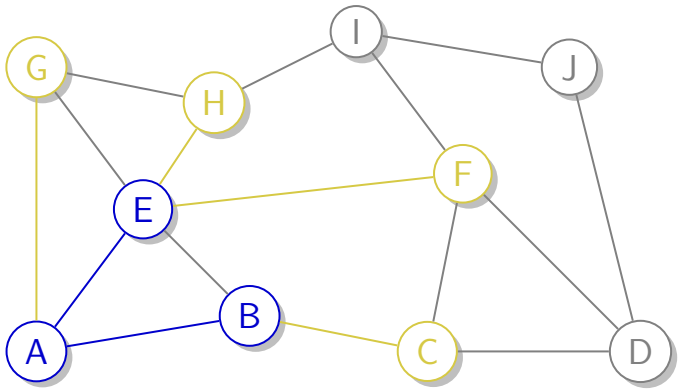
Průchod grafem do šířky, krok 13



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 |

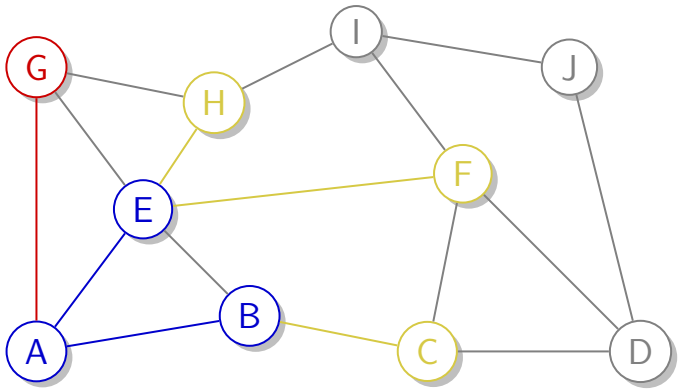
Průchod grafem do šířky, krok 14



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 |

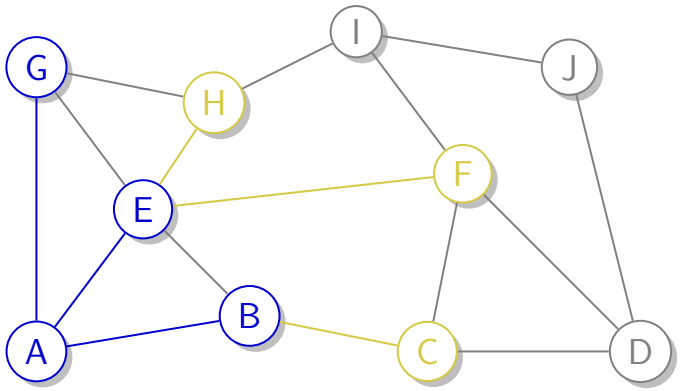
Průchod grafem do šířky, krok 15



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 | C | F | H |

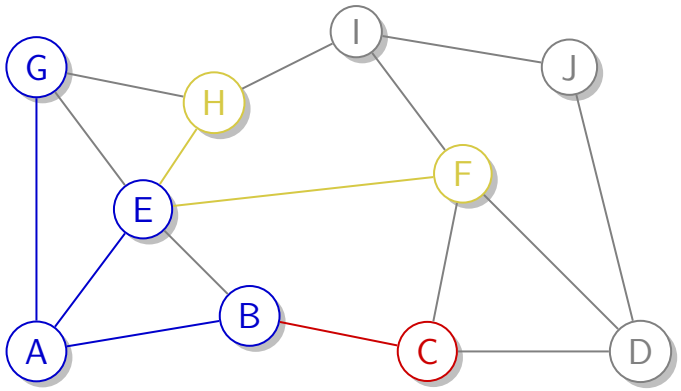
Průchod grafem do šířky, krok 16



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 | C | F | H |

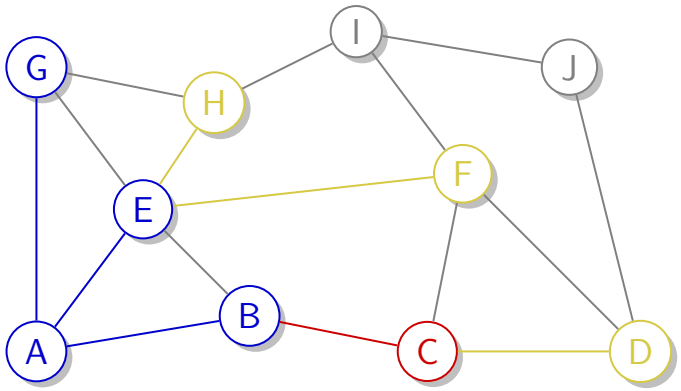
Průchod grafem do šířky, krok 17



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 | F | H |

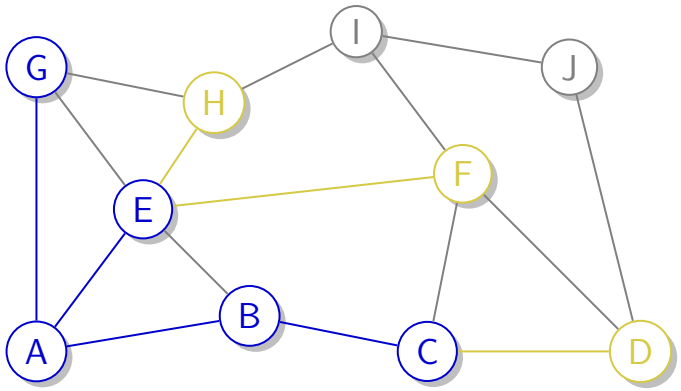
Průchod grafem do šířky, krok 18



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 | F | H | D |

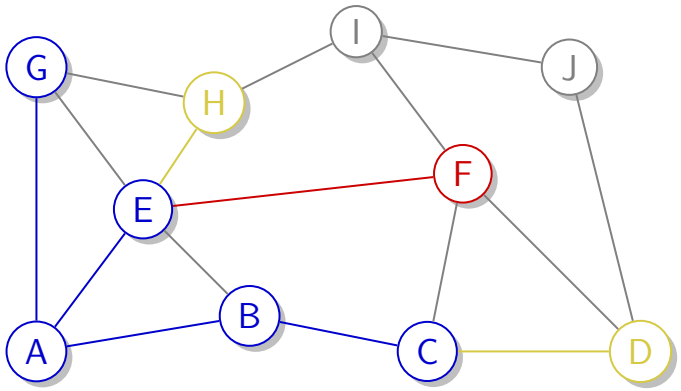
Průchod grafem do šířky, krok 19



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 | F | H | D |

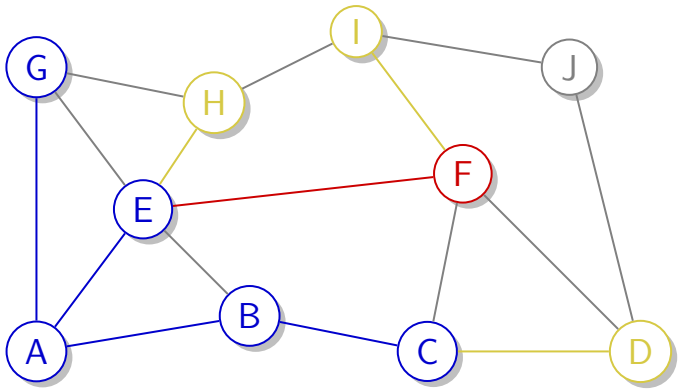
Průchod grafem do šířky, krok 20



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 |

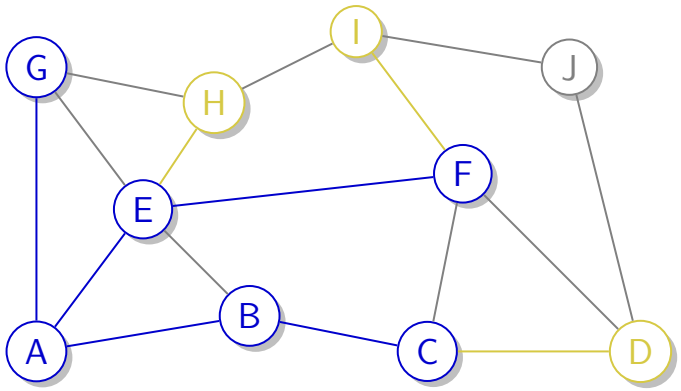
Průchod grafem do šířky, krok 21



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	∞	/

Q | H | D | I |

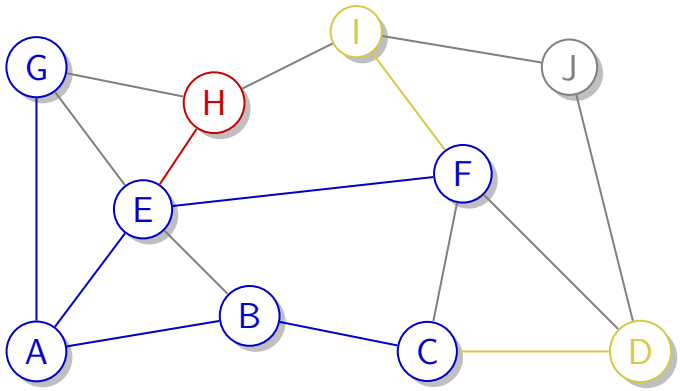
Průchod grafem do šířky, krok 22



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	∞	/

Q | H | D | I |

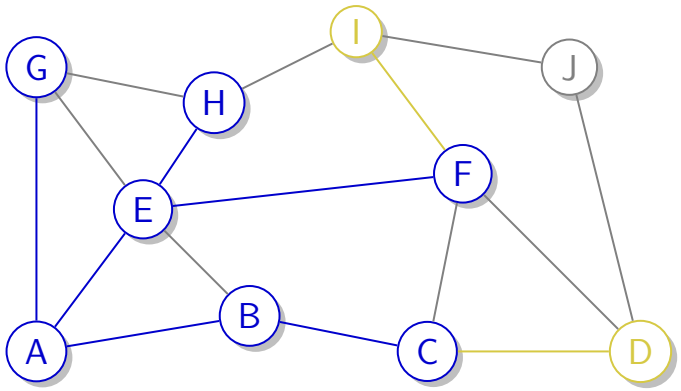
Průchod grafem do šířky, krok 23



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	∞	/

Q | D | I |

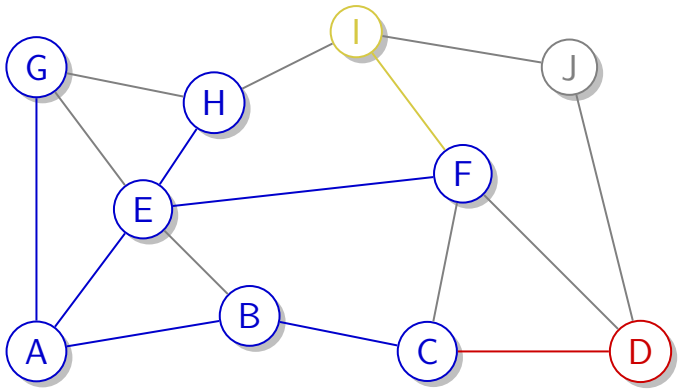
Průchod grafem do šířky, krok 24



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	∞	/

Q | D | I |

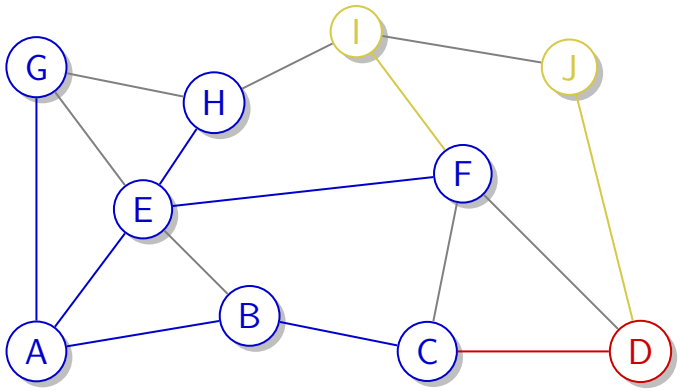
Průchod grafem do šířky, krok 25



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	∞	/

Q | I |

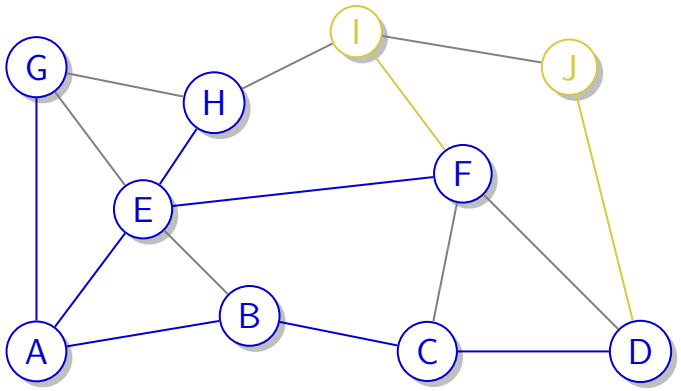
Průchod grafem do šířky, krok 26



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

Q | I | J |

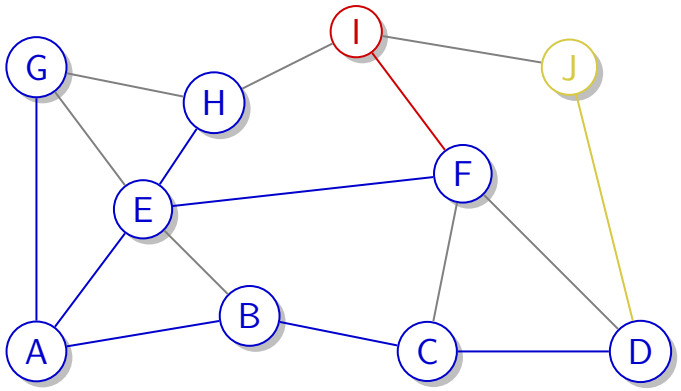
Průchod grafem do šířky, krok 27



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

Q | I | J |

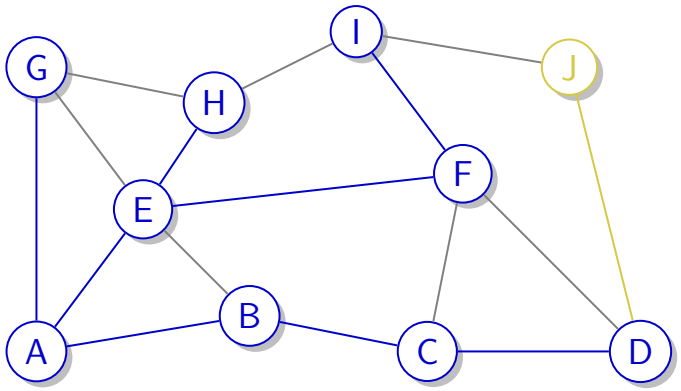
Průchod grafem do šířky, krok 28



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

Q | J |

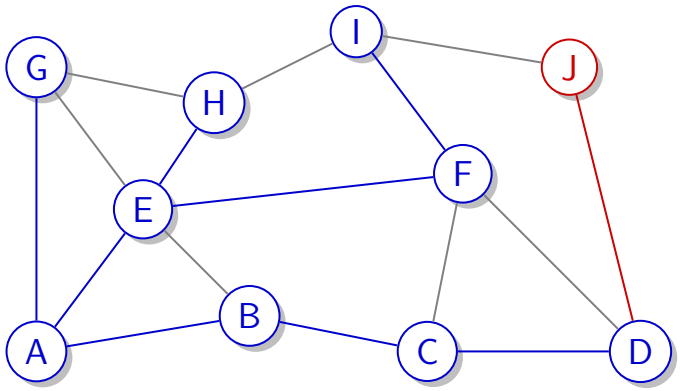
Průchod grafem do šířky, krok 29



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

Q | J |

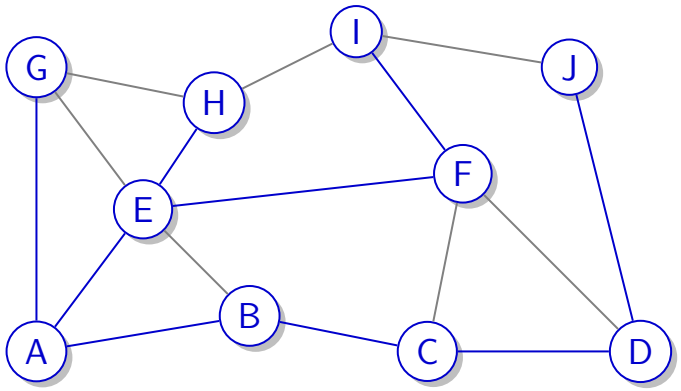
Průchod grafem do šířky, krok 30



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

Q

Průchod grafem do šířky, krok 31



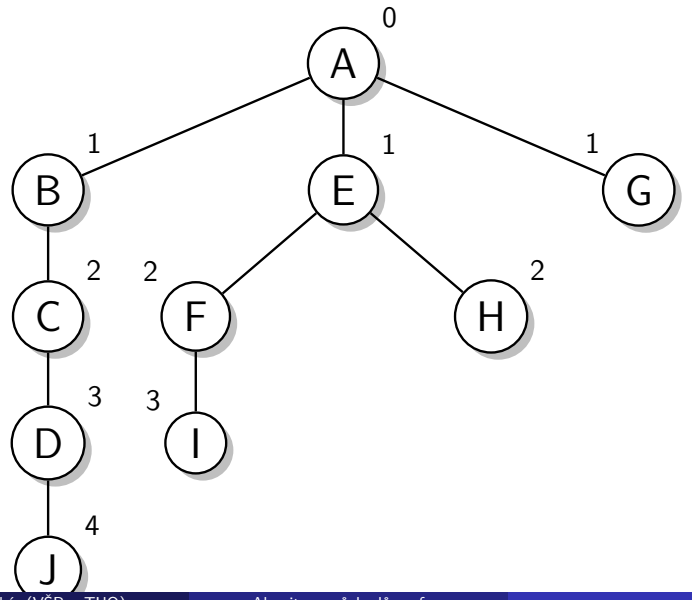
v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

Q

Průchod grafem do šířky

- Výstupem algoritmu je tzv. BF-strom.
- BF-strom je jednou z koster grafu.
- BF-strom obsahuje všechny vrcholy dostupné z počátečního vrcholu s.
- BF-strom je stromem nejkratších cest.
- Průměr souvislého grafu – maximální délka nejkratší cesty pro libovolné dva vrcholy.
- Složitost algoritmu: $O(|V| + |E|)$.

Průchod grafem do šířky – BF-strom



Prohledávání grafu do hloubky

- anglicky Depth-first Search (DFS)
- základem pro zjišťování počtu komponent grafu, vyhledání všech cest z daného vrcholu, testování acykličnosti grafu, topologické třídění.
- lze řešit rekurzivně, podobnost s backtrackingem.

Idea algoritmu

1. nejprve označkujeme vrchol s,
2. pak označkujeme jednoho následovníka vrcholu s, pak jeho následovníky atd.
3. až vyčerpáme všechny dostupné následovníky, vracíme se postupně zpět,
4. a pokračujeme dalším následovníkem vrcholu s.

Algoritmus lze přirovnat k prohledávání bludiště jedním člověkem.

Prohledávání grafu do hloubky – stavy vrcholu

- Každému vrcholu přiřadíme jeden ze čtyř možných stavů:
  - unknown – vrchol ještě nebyl navštíven, algoritmus k němu ještě nedošel.
  - discovered – vrchol byl již objeven, čeká v zásobníku na zpracování.
  - current – vrchol je zpracován, čeká v zásobníku na dokončení zpracování svých sousedů
  - finished – zpracování vrcholu a jeho sousedů bylo dokončeno.
- Každý vrchol je počátku ve stavu unknown, pak přejde do stavu discovered, current a nakonec se ocitne ve stavu finished.

Prohledávání grafu do hloubky

- Dále každému vrcholu v přiřadíme dvě časová razítka: $d[v]$ a $f[v]$.
- $d[v]$ označuje okamžik, kdy se vrchol dostal do stavu current.
- $f[v]$ označuje okamžik, kdy se vrchol dostal do stavu finished.
- Pro všechny vrcholy platí, že $d[v] < f[v]$.
- Před $d[v]$ byl vrchol ve stavu unknown nebo discovered, mezi $d[v]$ a $f[v]$ byl ve stavu current a po $f[v]$ je ve stavu finished.

Prohledávání grafu do hloubky

- Graf $G = (V, E)$
- Počáteční vrchol s, $s \in V$.
- Pomocné datové struktury:
  - *state* – tabulka se stavy všech vrcholů,
  - π – o každý vrchol u je v tabulce π uložen jeho předchůdce, pokud neexistuje obsahuje tabulka hodnotu nothing,
  - d – tabulka časových razítek s časy přechodu vrcholu do stavu current,
  - f – tabulka časových razítek s časy přechodu vrcholu do stavu finished,
  - *time* – globální hodiny logického času.
  - *stack* – zásobník vrcholů ve stavu discovered nebo current.

Algoritmus průchodu grafem do hloubky

```
   Input  : Graf G(V, E), počáteční vrchol s ∈ V
   Output: DF-strom
42 Init(V);
43 state [s] ← discovered;
44 stack ← ∅;
45 Push(stack, s);
46 time ← 0;
47 while stack ≠ ∅ do
48 |  u ← Top(stack);
49 |  switch state [u] do
50 |  |  case discovered do
51 |  |  |  ProcessDiscoveredVertex(u);
52 |  |  end
53 |  |  case current do
54 |  |  |  ProcessCurrentVertex(u);
55 |  |  end
56 |  end
57 end
```


Algoritmus průchodu grafem do hloubky (pokrač.)

```
1 Procedure Init(V)
  |  Input  : Množina vrcholů V
2 |  foreach u ∈ V do
3 |  |  state [u] ← unknown;
4 |  |  d [u] ← f [u] ← ∞;
5 |  |  π [u] ← nothing;
6 |  end
7 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

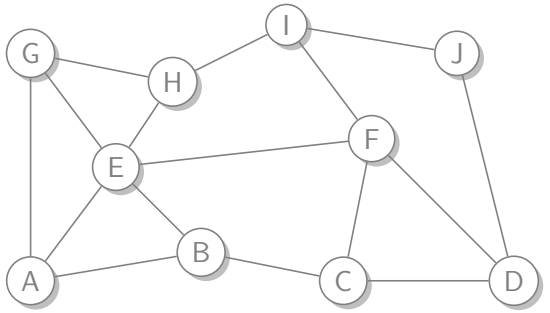
Vrcholy grafu:

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

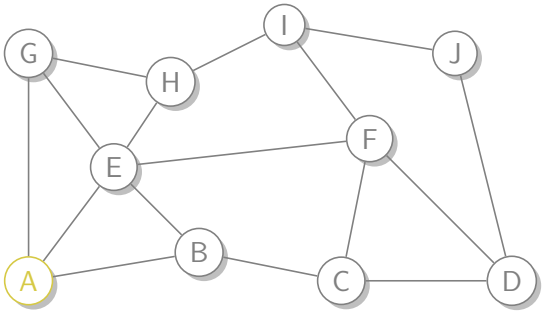


v	$d[v]$	$f[v]$	$\pi[v]$
A	∞	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

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

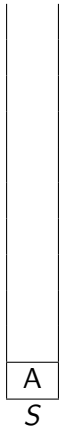


Průchod grafem do hloubky, krok 2

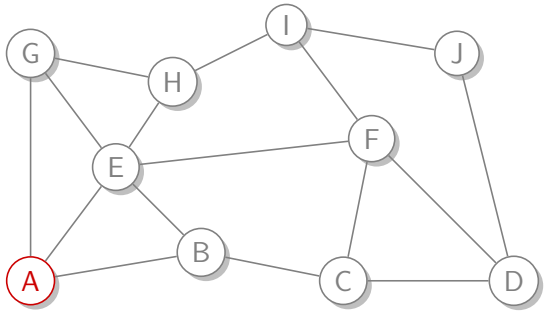


v	$d[v]$	$f[v]$	$\pi[v]$
A	∞	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

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

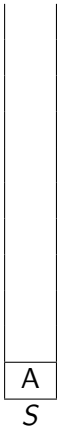


Průchod grafem do hloubky, krok 3

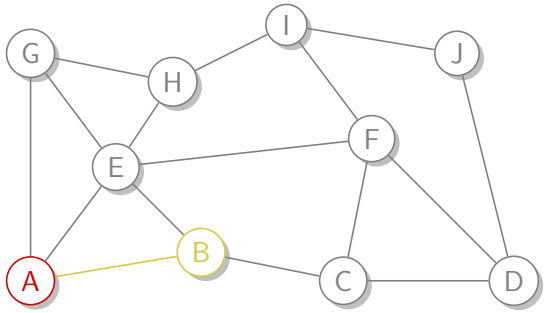


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

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

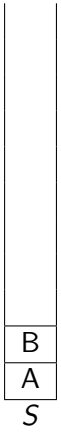


Průchod grafem do hloubky, krok 4

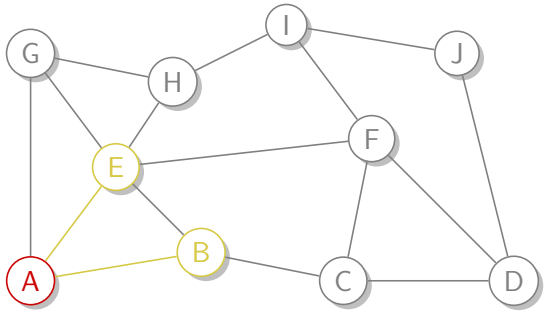


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

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

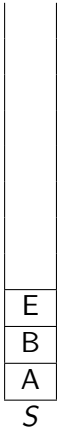


Průchod grafem do hloubky, krok 5

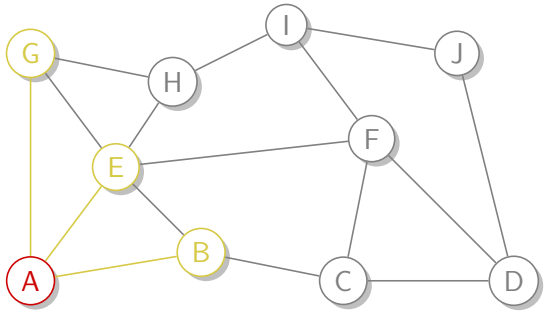


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

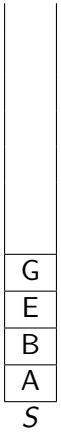


Průchod grafem do hloubky, krok 6

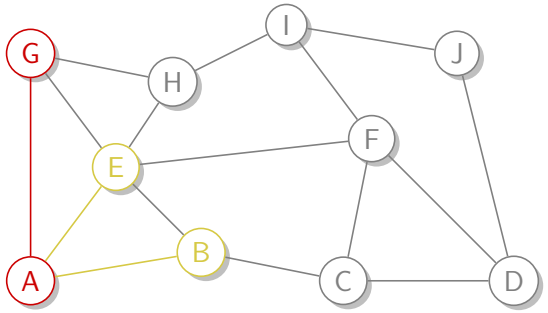


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

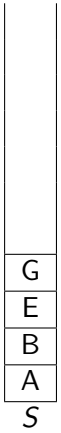


Průchod grafem do hloubky, krok 7

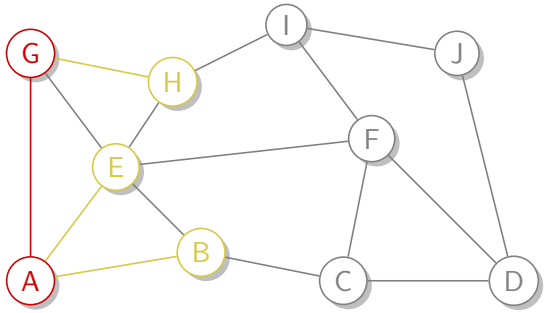


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

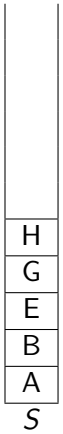


Průchod grafem do hloubky, krok 8

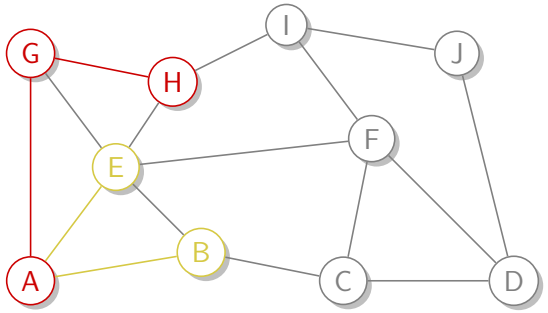


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

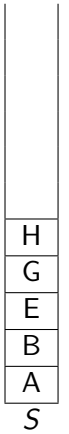


Průchod grafem do hloubky, krok 9

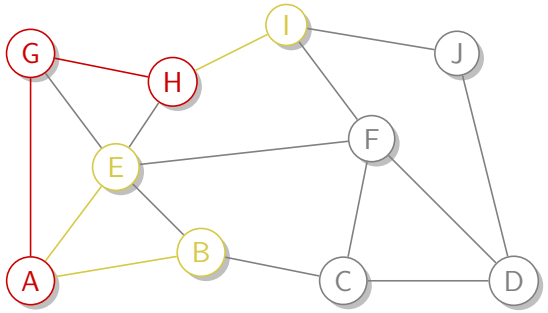


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

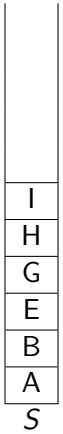


Průchod grafem do hloubky, krok 10

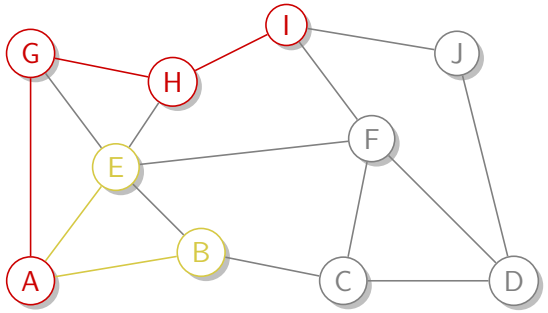


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

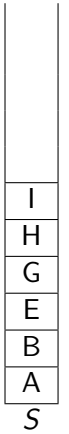


Průchod grafem do hloubky, krok 11

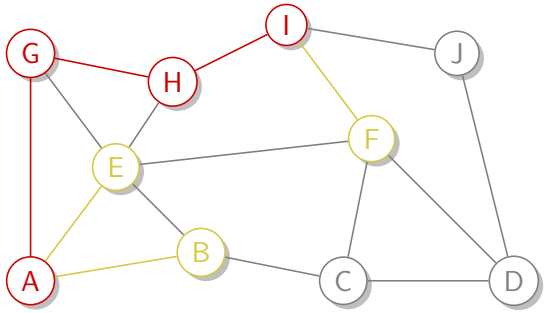


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

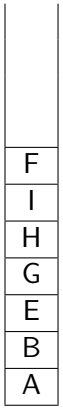


Průchod grafem do hloubky, krok 12

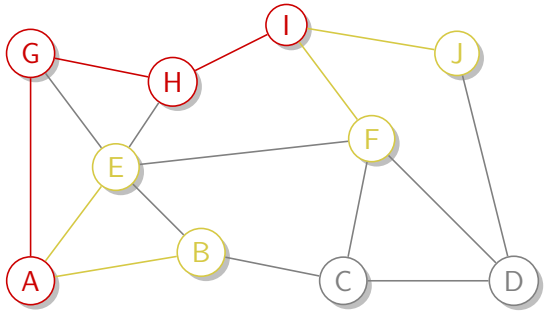


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

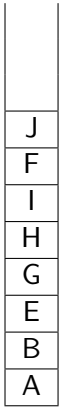


Průchod grafem do hloubky, krok 13

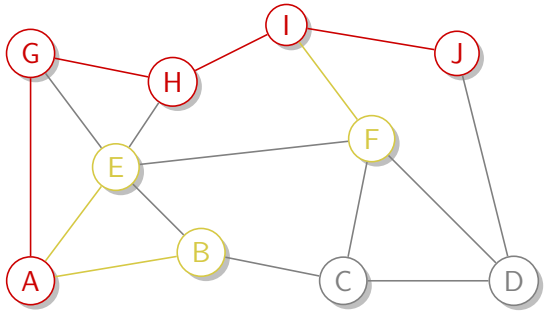


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

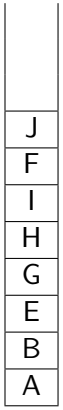


Průchod grafem do hloubky, krok 14

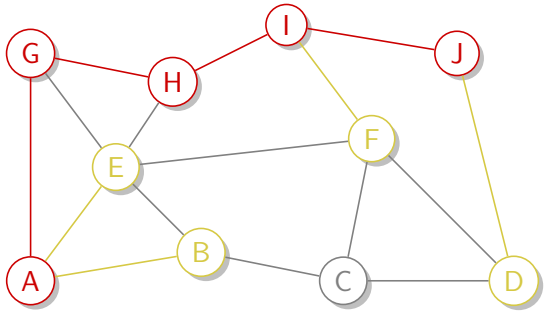


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

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

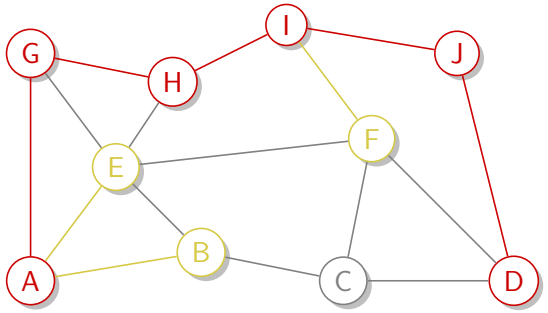


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	J
E	∞	∞	A

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



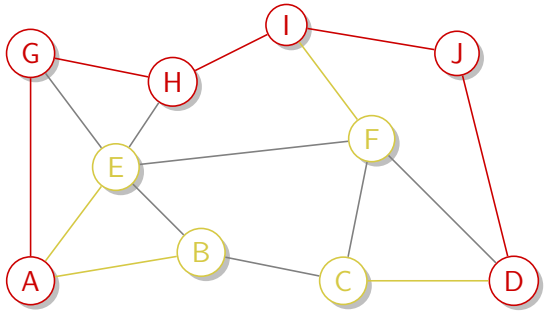
v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	6	∞	J
E	∞	∞	A

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

D
J
F
I
H
G
E
B
A

S

Průchod grafem do hloubky, krok 17



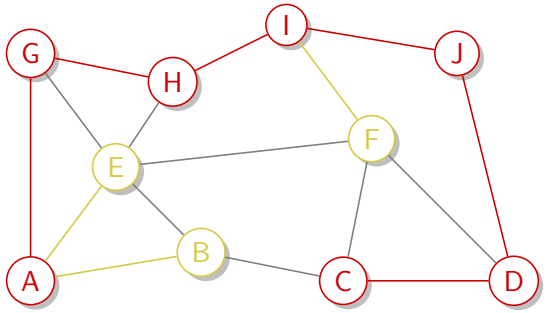
v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	D
D	6	∞	J
E	∞	∞	A

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

C
D
J
F
I
H
G
E
B
A

S

Průchod grafem do hloubky, krok 18



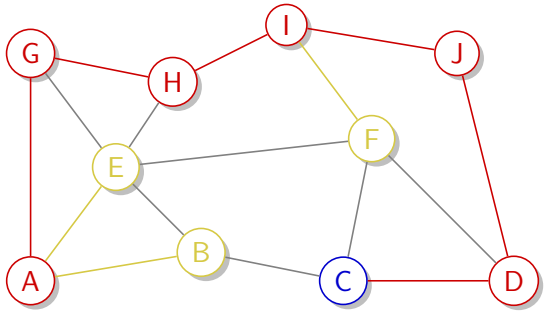
v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	∞	D
D	6	∞	J
E	∞	∞	A

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

C
D
J
F
I
H
G
E
B
A

S

Průchod grafem do hloubky, krok 19



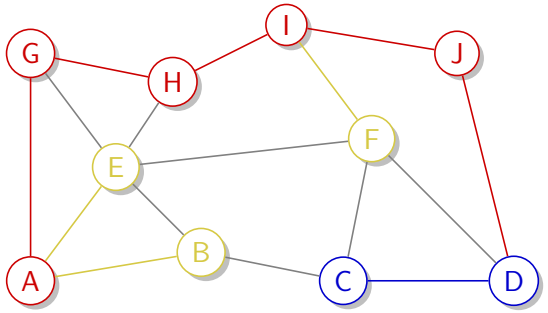
v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	∞	J
E	∞	∞	A

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

D
J
F
I
H
G
E
B
A

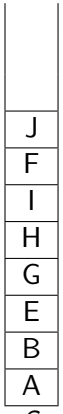
S

Průchod grafem do hloubky, krok 20

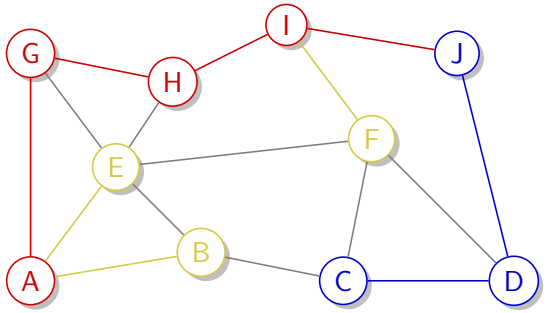


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

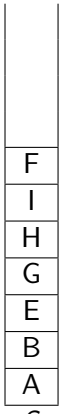


Průchod grafem do hloubky, krok 21

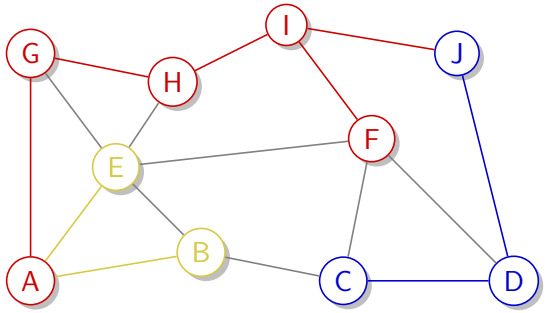


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

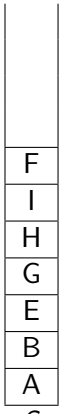


Průchod grafem do hloubky, krok 22

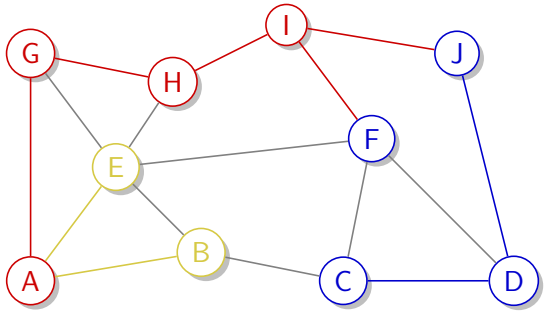


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

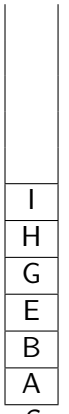


Průchod grafem do hloubky, krok 23

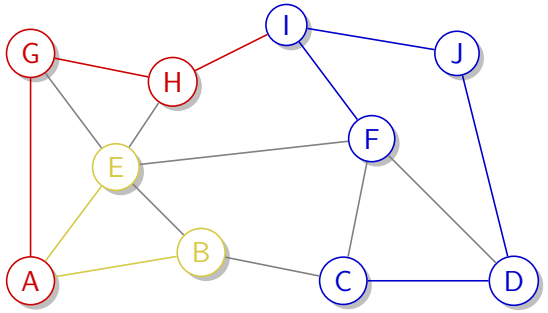


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

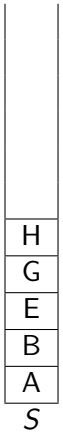


Průchod grafem do hloubky, krok 24

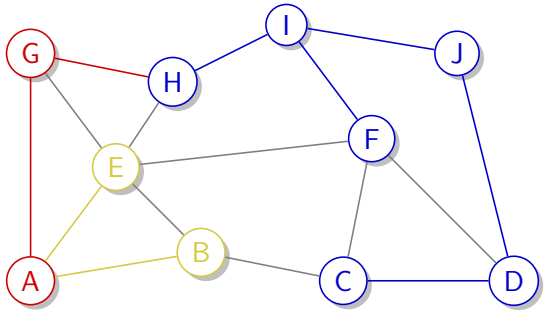


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

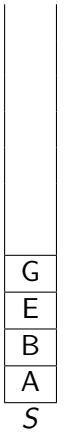


Průchod grafem do hloubky, krok 25

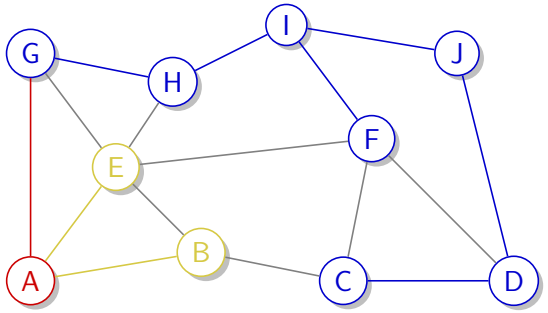


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

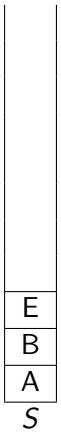


Průchod grafem do hloubky, krok 26

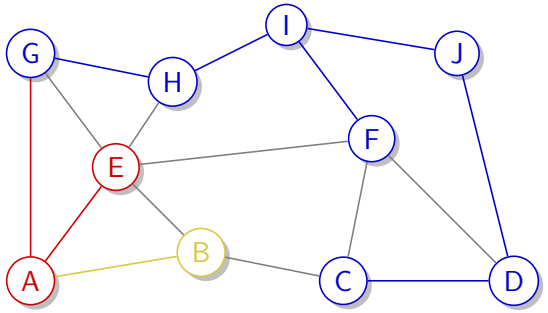


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

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

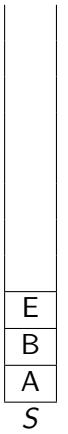


Průchod grafem do hloubky, krok 27

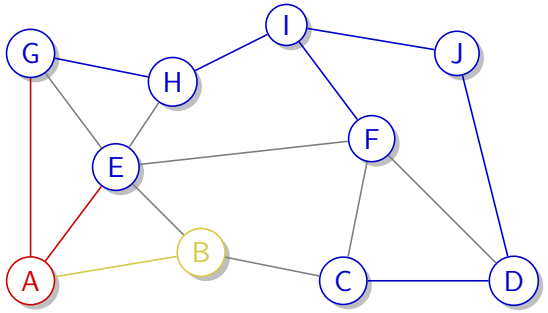


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	16	∞	A

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

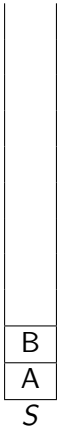


Průchod grafem do hloubky, krok 28

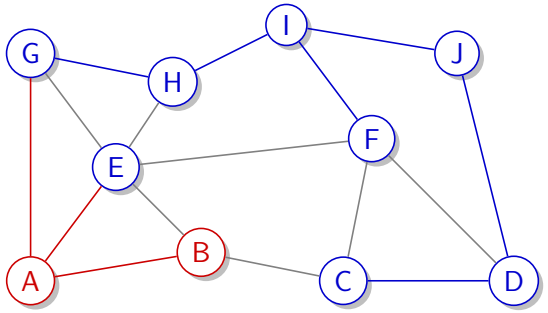


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	16	17	A

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

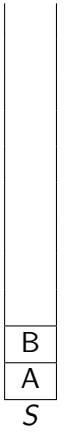


Průchod grafem do hloubky, krok 29

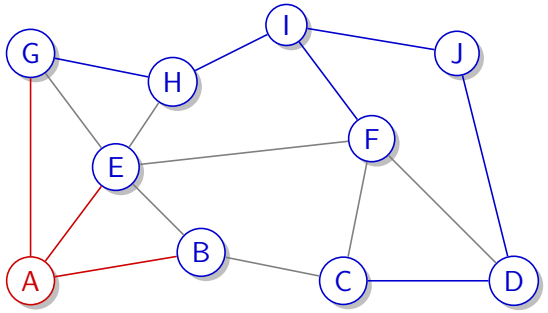


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	18	∞	A
C	7	8	D
D	6	9	J
E	16	17	A

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

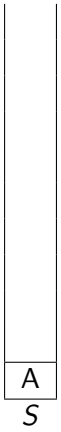


Průchod grafem do hloubky, krok 30

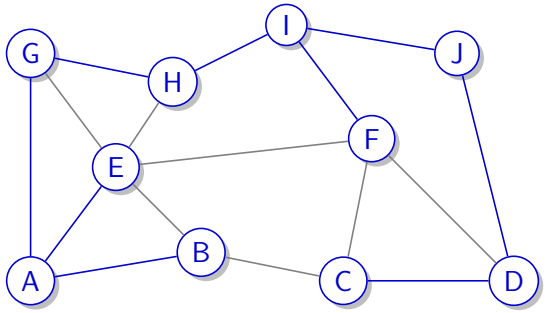


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	18	19	A
C	7	8	D
D	6	9	J
E	16	17	A

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

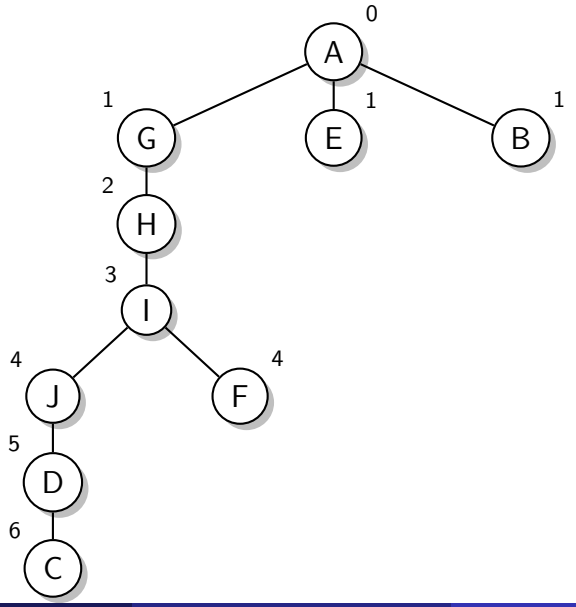
v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I



Průchod grafem do hloubky

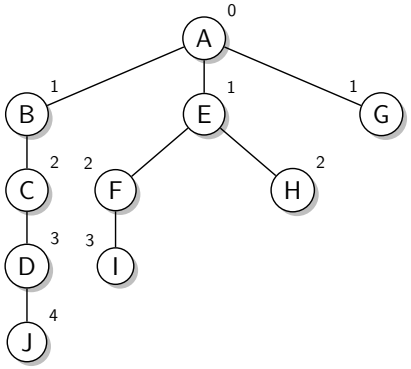
- Výstupem algoritmu je tzv. DF-strom, jedna z koster grafu.
- Závorková struktura stromu – Mějme vrchol u, pro všechny vrcholy v v podstromu s kořenem u platí, že $d[u] < d[v] < f[v] < f[u]$.
- Lze jednoduše zjišťovat vztah rodič–potomek. Indexování XML dokumentů.
- Klasifikace hran: dopředné (součást DF-stromu), zpětné (hrany mimo DF-strom)
- Složitost algoritmu: $O(|V| + |E|)$.

Průchod grafem do hloubky – DF-strom

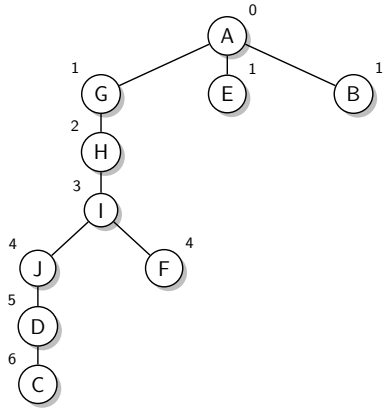


Srovnání průchodů grafem

Průchod do šířky



Průchod do hloubky

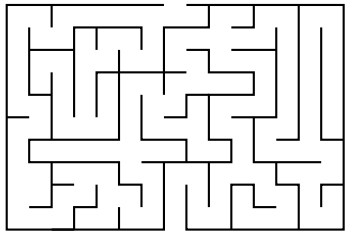


Aplikace

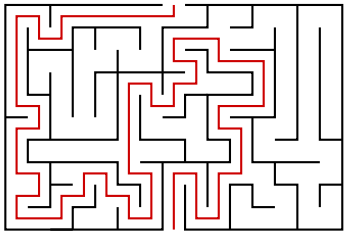
- Hledání cesty v bludišti
- Počítačová grafika – vyplňování oblasti

Hledání cesty v bludišti

Bludiště



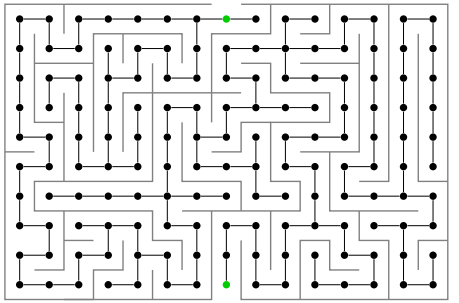
Nalezená cesta bludištěm



Zadání úlohy

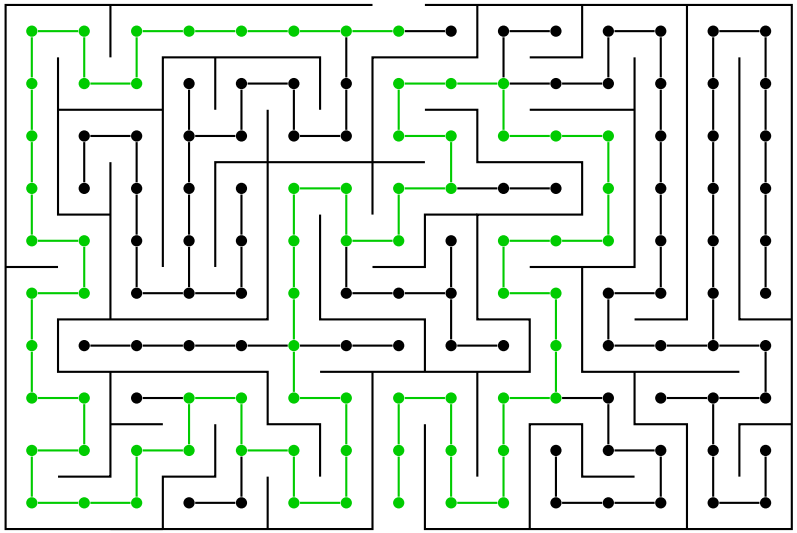
Máme najít cestu daným bludištěm od daného vchodu k danému východu nebo dospět k závěru, že cesta neexistuje.

Bludiště jako graf



- Bludiště převedeme na graf:
  - vrcholy
  - hrany
- Zvolíme některý z algoritmů průchodu grafem
  1. označíme si vchod a východ
  2. provedeme algoritmus
  3. pomocí odkazů π na předchůdce zrekonstruujeme cestu od východu ke vchodu

Hledání cesty v bludišti – řešení



Animace průchodu grafem do šířky – legenda

Vrcholy grafu:

šedá	dosud neprozkoumaný vrchol
žlutá	vrchol čekající ve frontě na zpracování
červená	právě zpracovávaný vrchol
modrá	zpracovaný vrchol
zelená	vrchol na cestě mezi vchodem a východem z bludiště
n	vzdálenost (počet hran) od vchodu do bludiště

Hrany grafu:

šedá	hrana mezi neprozkoumanými vrcholy
žlutá	hrana incidentní s čekajícím vrcholem
červená	hrana incidentní s právě zpracovávaným vrcholem
modrá	hrana mezi zpracovanými vrcholy
zelená	hrana na cestě mezi vchodem a východem z bludiště
$u \to v$	$\pi[u] = v$

Animace průchodu grafem do šířky

Animace jsou v poznámkách k prezentaci vynechány.

Animace průchodu grafem do šířky

Animace jsou v poznámkách k prezentaci vynechány.

Animace průchodu grafem do hloubky – legenda

Vrcholy grafu:

šedá	dosud neprozkoumaný vrchol
žlutá	vrchol čekající ve frontě na zpracování
červená	právě zpracovávaný vrchol
modrá	zpracovaný vrchol
zelená	vrchol na cestě mezi vchodem a východem z bludiště
$\frac{d[u]}{f[u]}$	logický čas navštívení a dokončení průchodu vrcholem u

Hrany grafu:

šedá	hrana mezi neprozkoumanými vrcholy
žlutá	hrana incidentní s čekajícím vrcholem
červená	hrana incidentní s právě zpracovávaným vrcholem
modrá	hrana mezi zpracovanými vrcholy
zelená	hrana na cestě mezi vchodem a východem z bludiště
$u \rightarrow v$	$\pi[u] = v$

Animace průchodu grafem do hloubky

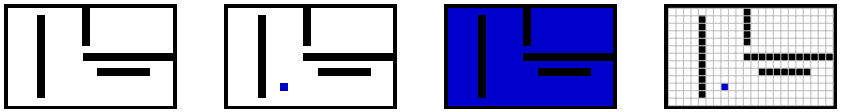
Animace jsou v poznámkách k prezentaci vynechány.

Animace průchodu grafem do hloubky

Animace jsou v poznámkách k prezentaci vynechány.

Počítačová grafika – vyplňování oblasti

- Zpracování rastrové grafiky – obrázek je tvořen pravoúhlou mřížkou pixelů.
- Oblast je ohraničena zvolenou barvou, například černou.
- Myší „ukážeme“ do zadané oblasti.
- Máme za úkol vyplnit zadanou oblast zvolenou barvou.



Příklady použití:

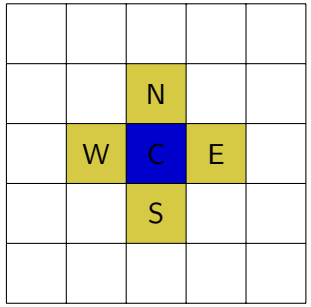
- interaktivní práce s grafickým editorem, úprava fotek atd.
- rasterizace vektorové grafiky, např. písmena v OpenType fontech jsou popsána obrysy, na tiskárně pak dochází k rozložení na pixely podle daného DPI a vyplnění obrysů.

Počítačová grafika – vyplňování oblasti

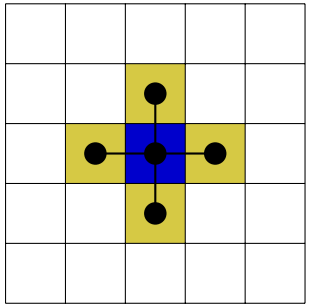
Řešení

1. převod rastru pixelů na graf – definice vrcholů a hran,
2. využití algoritmu průchodu grafem.

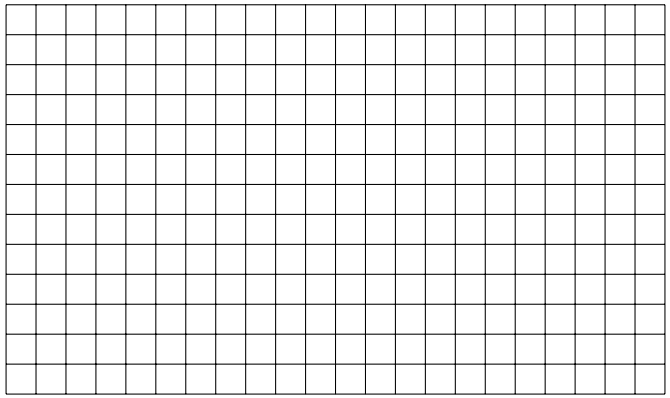
Rastr pixelů



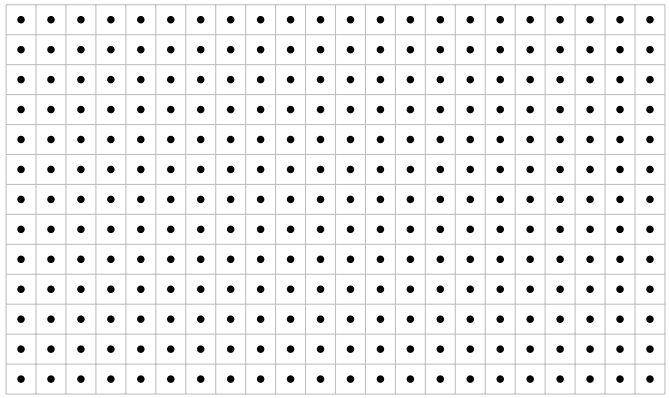
Grafová reprezentace



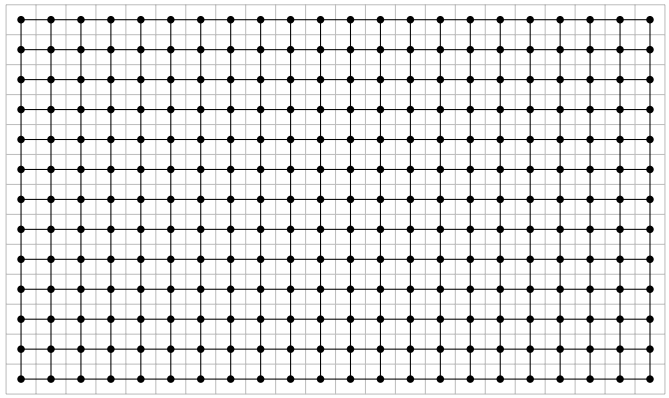
Počítačová grafika – vyplňování oblasti



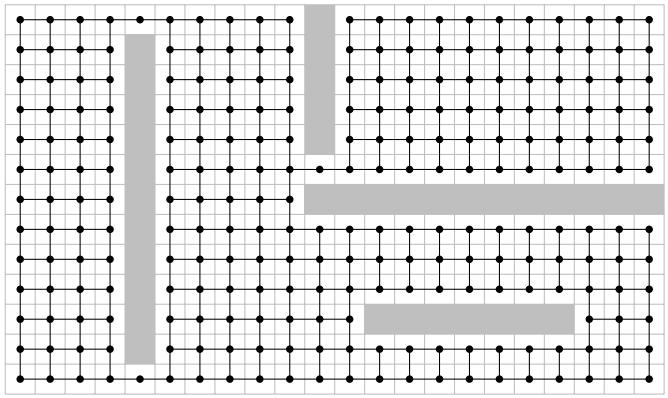
Počítačová grafika – vyplňování oblasti



Počítačová grafika – vyplňování oblasti



Počítačová grafika – vyplňování oblasti



Počítačová grafika – vyplňování oblasti

Varianty řešení:

- algoritmus průchodu grafem:
  - průchod do šířky, pixely čekající na zpracování jsou ukládány do fronty,
  - průchod do hloubky, pixely čekající na zpracování jsou ukládány do zásobníku a
  - náhodný – pixely čekající na zpracování jsou ukládány do pomocné datové struktury na náhodné místo a z náhodného místa jsou vybírány.
- pořadí zpracování sousedů aktuálního pixelu:
  - N E S W,
  - N S E W,
  - náhodně.

Počítačová grafika – vyplňování oblasti

Legenda k animacím:

- algoritmy průchodu:
  - BFS – průchod do šířky,
  - DFS – průchod do hloubky,
  - Random – náhodný průchod.
- pořadí zpracování sousedů
  - N E S W,
  - náhodně.
- barva pixelů vyplňované oblasti:
  - bílá – dosud neprozkoumaný pixel,
  - žlutá – pixel čekající na zpracování,
  - červená – právě zpracovávaný pixel,
  - modrá – zpracovaný, vybarvený, pixel.
- Měřítko pod animací zobrazuje aktuální a maximální počet pixelů čekajících na zpracování v pomocné datové struktuře.

Animace vyplňování oblasti – BFS, NESW

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – BFS, NESW

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – DFS, NESW

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – DFS, NESW

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – BFS, Random

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – DFS, Random

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – DFS, Random

Animace jsou v poznámkách k prezentaci vynechány.

Animace vyplňování oblasti – Random, Random

Animace jsou v poznámkách k prezentaci vynechány.

Kontrolní otázky

1. Matice sousednosti grafu. Jaký je řád této matice? Je čtvercová? Jaký je význam prvků v matici?
2. Matice sousednosti grafu. Jak poznám, že graf je neorientovaný? Jak poznám, že graf neobsahuje smyčky?
3. Reprezentace grafu pomocí seznamu sousedních vrcholů. Jaké jsou výhody této implementace oproti matici sousednosti? Zaměřte se na operace jako je vložení nového vrcholu do grafu či paměťovou složitost obou reprezentací grafu.
4. Co je cílem algoritmů průchodu grafem?
5. Jaké tři algoritmy průchodu grafem znáte?
6. Popište obecný algoritmus značkování vrcholů.
7. V čem spočívá nevýhoda obecného značkovacího algoritmu? Co v jeho implementaci může zvyšovat časovou složitost?
8. Popište myšlenku algoritmu průchodu grafem do šířky (BFS).

Kontrolní otázky (pokrač.)

9. V algoritmu BFS se používají pro označení vrcholů tři barvy – bílá, šedá a černá. Jaký význam mají tato tři označení?
10. V jaké datové struktuře je při průchodu BFS udržována množina šedých vrcholů?
11. Jaký význam mají hodnoty $d[u]$ a $\pi[u]$ z algoritmu BFS?
12. Jak detekuji konec BFS algoritmu? Jak poznám, že jsem skončil?
13. Výstupem BFS je tzv. BF-strom. Jaké vrcholy tento strom obsahuje?
14. Mám dva vrcholy u a v v BF-stromu. Jaká je délka cesty mezi těmito dvěma vrcholy ze všech možných cest mezi těmito vrcholy v původním grafu? Nejkratší? Nejdelší? Průměrná?
15. Co je to průměr grafu?
16. Jaká je složitost BFS algoritmu?
17. Popište myšlenku algoritmu průchodu grafem do hloubky (DFS).

Kontrolní otázky (pokrač.)

18. DFS algoritmus přiřazuje každému vrcholu dvě tzv. časová razítka. Jaký mají význam?
19. Jak mohu, pomocí časových razítek, určit pro dva vrcholy vztah rodič-potomek v DF-stromu?
20. Jaká je složitost DFS algoritmu?
21. DFS lze implementovat i nerekurzivně? Pomocí jaké datové struktury mohu odstranit rekurzi?
22. Co je to kostra grafu?
23. Co je to dopředná a zpětná hrana?
24. Co je to topologické třídění?
25. Jak mohu detekovat v grafu cyklus?

Děkuji za pozornost

Binární vyhledávací stromy I

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Binární strom

Definice

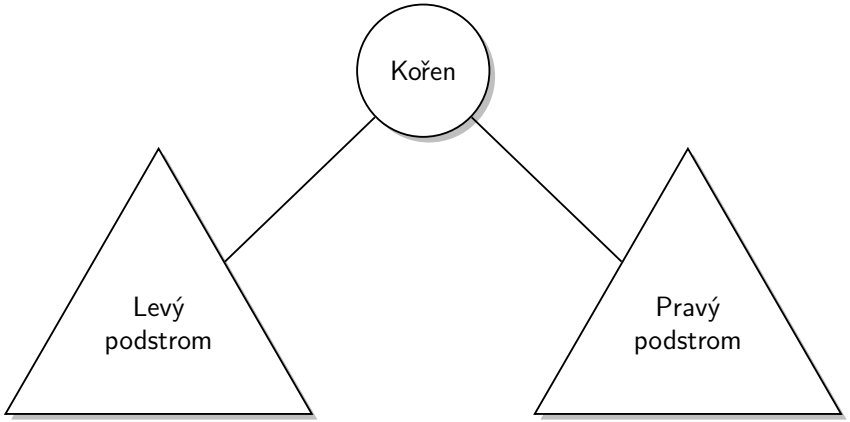
Binární strom je struktura definovaná nad konečnou množnou uzlů, která:

Pravidlo 1 – neobsahuje žádný uzel nebo

Pravidlo 2 – je složena ze tří disjunktních množin uzlů:

- kořene stromu,
- binárního stromu zvaného levý podstrom,
- a binárního stromu zvaného pravý podstrom.

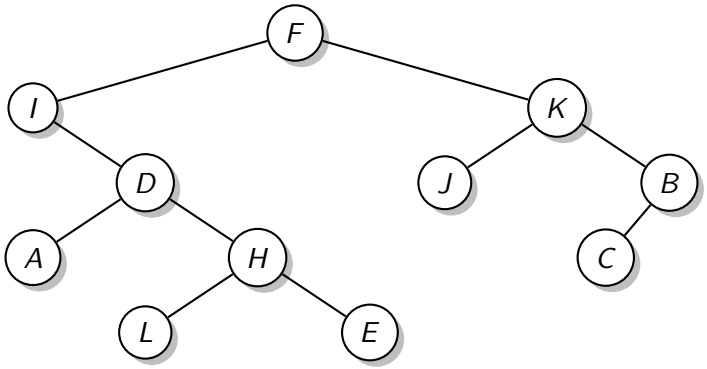
Binární strom – grafické znázornění rekurzivní definice



Binární vs. seřazený strom

- Binární strom není seřazený strom, kde má každý uzel stupeň nejvýše dva.
- Pokud má uzel jednoho potomka, je rozdíl zda je to potomek levý nebo pravý důležitý.
- V seřazeném stromu není možné rozlišit zda je jediný potomek levý nebo pravý, prostě je jeden.

Binární strom, příklad



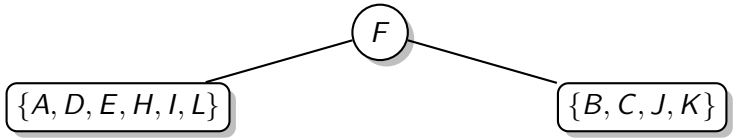
Budování binárního stromu podle rekurzivní definice

- Výchozí množina uzlů: $\{A, B, C, D, E, F, H, I, J, K, L\}$
- 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.

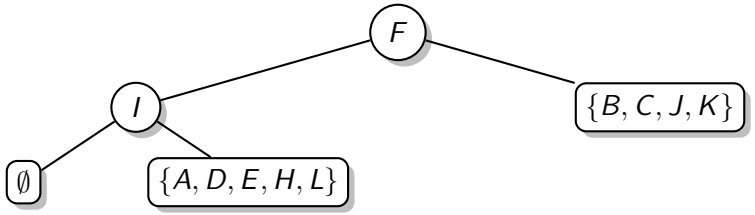
Budování binárního stromu podle rekurzivní definice

$\{A, B, C, D, E, F, H, I, J, K, L\}$

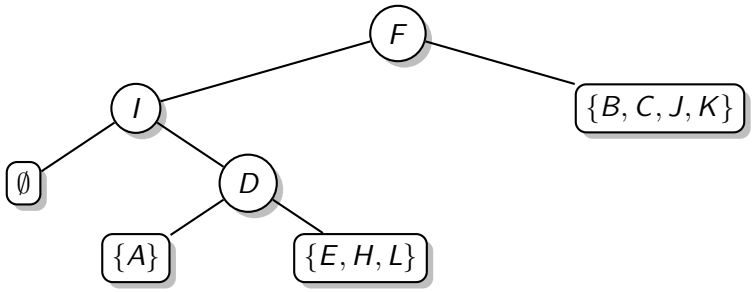
Budování binárního stromu podle rekurzivní definice



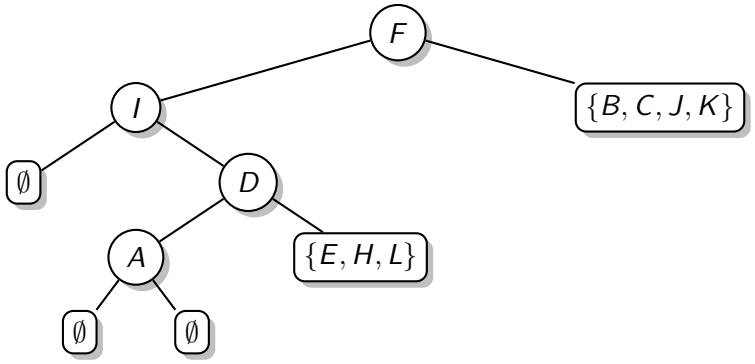
Budování binárního stromu podle rekurzivní definice



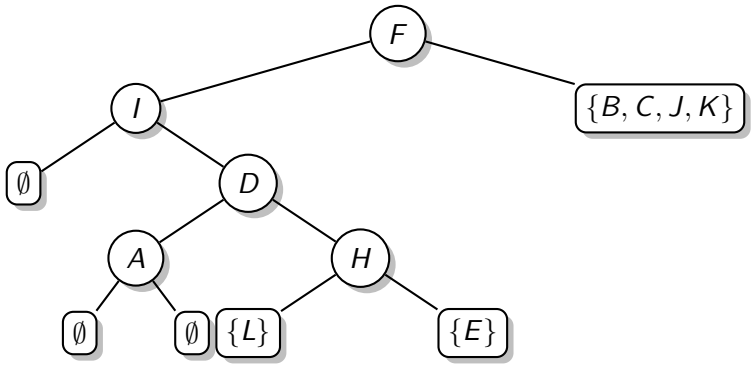
Budování binárního stromu podle rekurzivní definice



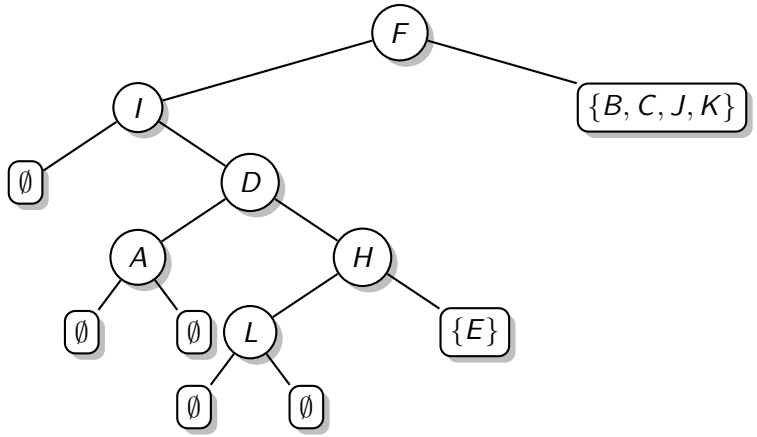
Budování binárního stromu podle rekurzivní definice



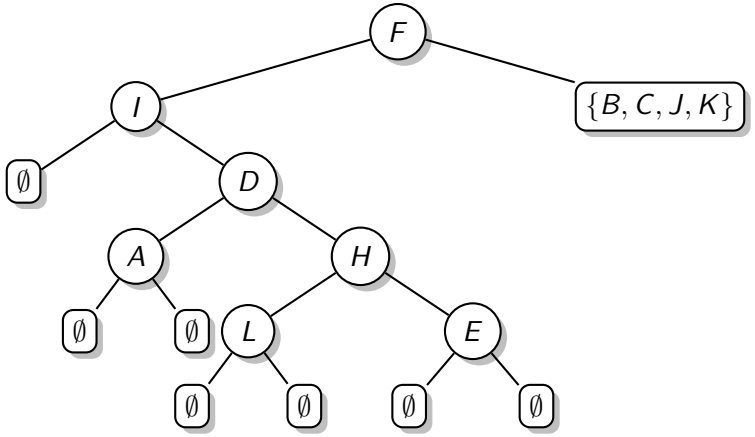
Budování binárního stromu podle rekurzivní definice



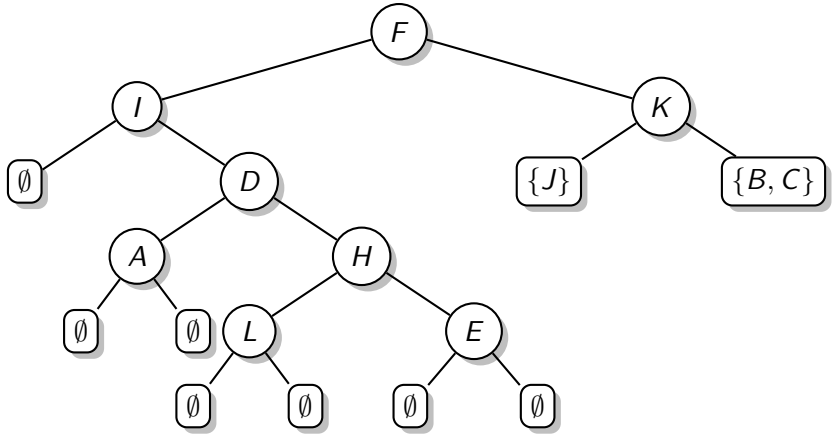
Budování binárního stromu podle rekurzivní definice



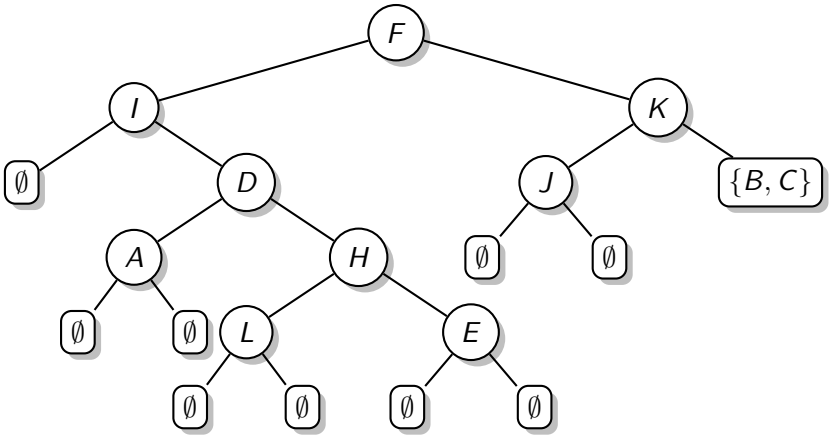
Budování binárního stromu podle rekurzivní definice



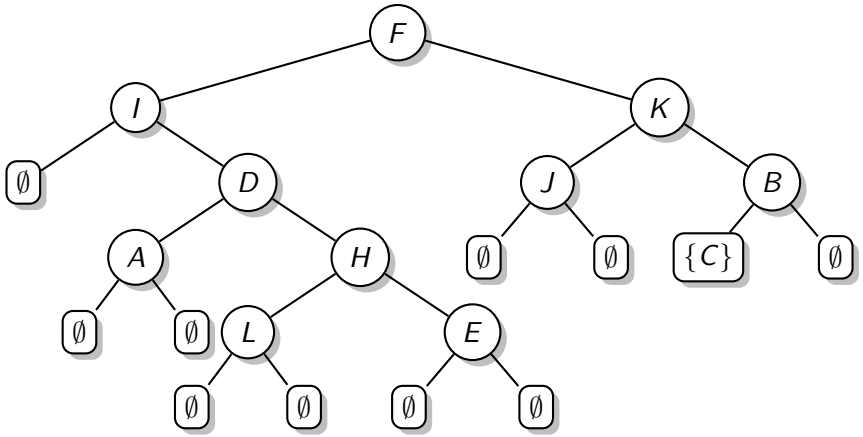
Budování binárního stromu podle rekurzivní definice



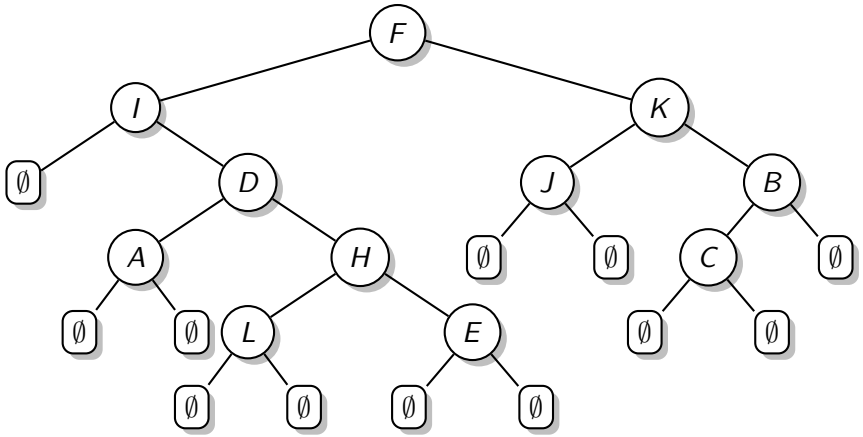
Budování binárního stromu podle rekurzivní definice



Budování binárního stromu podle rekurzivní definice

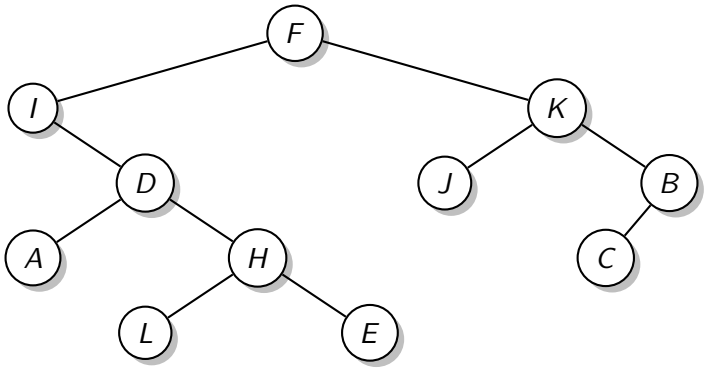


Budování binárního stromu podle rekurzivní definice

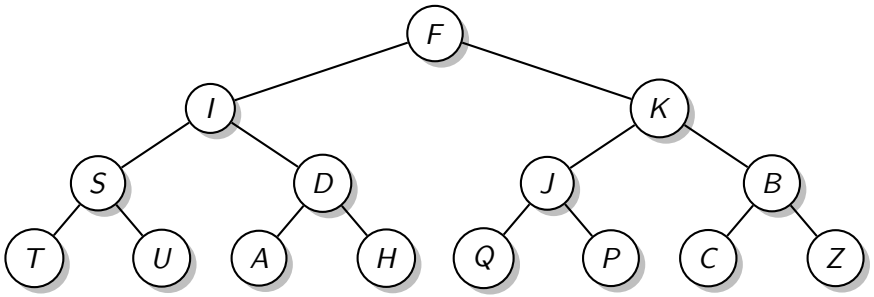


Budování binárního stromu podle rekurzivní definice

Odstraníme-li vizualizaci prázdných podstromů, zobrazených jako prázdné množiny, dostáváme výsledný binární strom v obvyklé podobě.

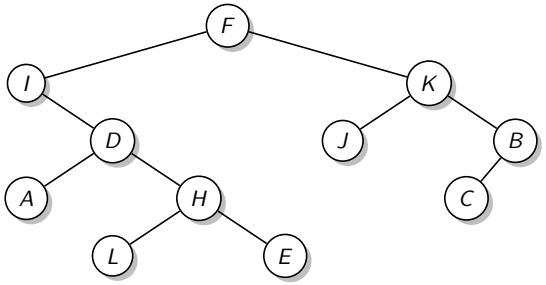


Úplný binární strom



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).

Poznámky

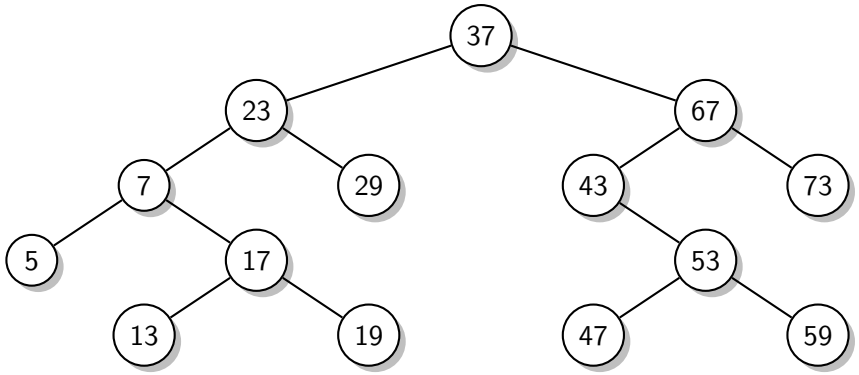
- Navigační pravidlo tedy určuje, jak mají být data v binárním vyhledávacím stromu rozmístěna.
- Znalosti rozmístění dat ve stromu využijeme při jejich vyhledávání.
- Algoritmy pro vložení a vyjmutí ze stromu jsou navázány na algoritmus vyhledávání.
- Binární vyhledávací strom je tedy už od počátku budován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

Binární vyhledávací strom – vyhledávání

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

1. Strom s kořenem r je prázdný ($r = NULL$), potom tento strom nemůže obsahovat uzel s klíčem k a hledání končí neúspěchem.
2. V opačném případě srovnáme klíč k s klíčem kořene právě zkoumaného stromu resp. jeho podstromu r. V případě, že
   - 2.1 $k = \mathrm{klic}(r)$ strom obsahuje uzel s klíčem k. Hledání končí úspěšně;
   - 2.2 $k < \mathrm{klic}(r)$ vzhledem k vlastnostem binárních vyhledávacích stromů, jsou všechny uzly s klíči menšími než je klíč r v jeho levém podstromu, pokračujeme rekurzivně v levém podstromu;
   - 2.3 $k > \mathrm{klic}(r)$ na rozdíl od předchozího případu jsou všechny uzly s klíči většími než je klíč r v pravém podstromu, pokračujeme pravým podstromem.

Binární vyhledávací strom



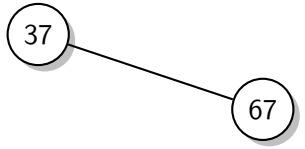
Binární vyhledávací strom – vkládání

- Vložení uzlu (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 (klíč) je připojen jako nový list ke stromu – strom roste prostřednictvím listů.
- Otázkou je co s duplicitami? Řešení závisí na povaze konkrétní řešené úlohy.

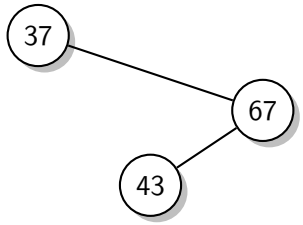
Binární vyhledávací strom – vložení klíče 37



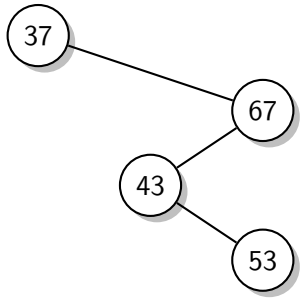
Binární vyhledávací strom – vložení klíče 67



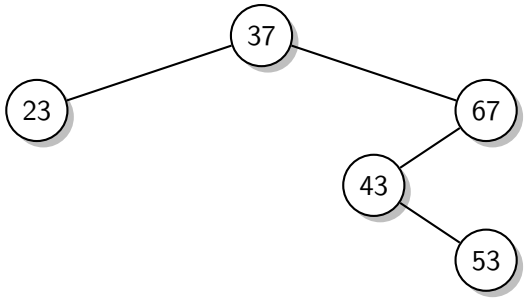
Binární vyhledávací strom – vložení klíče 43



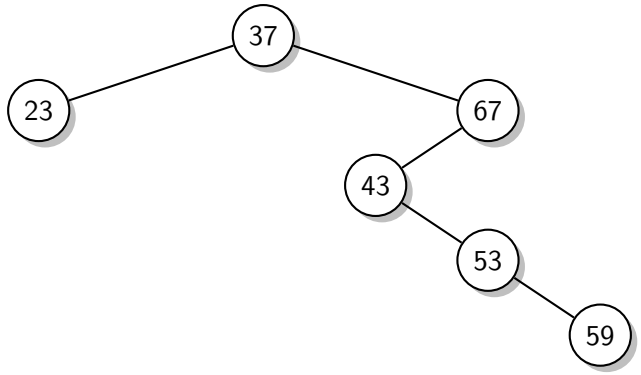
Binární vyhledávací strom – vložení klíče 53



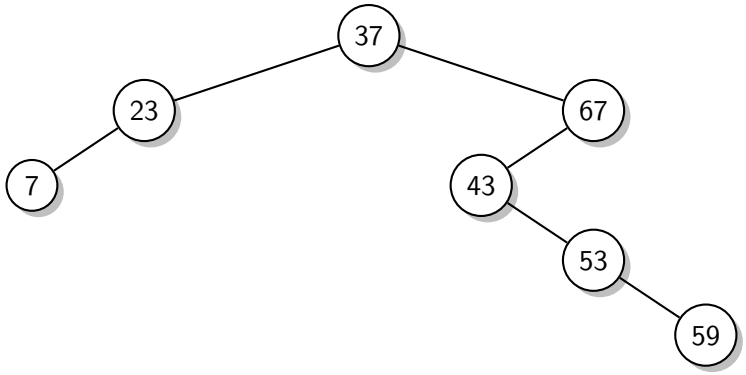
Binární vyhledávací strom – vložení klíče 23



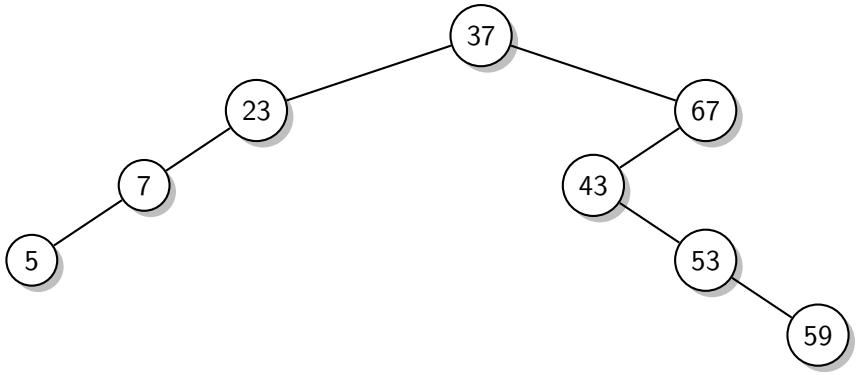
Binární vyhledávací strom – vložení klíče 59



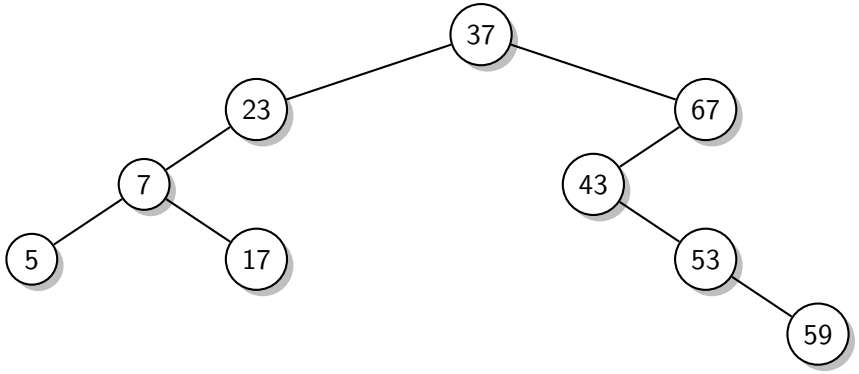
Binární vyhledávací strom – vložení klíče 7



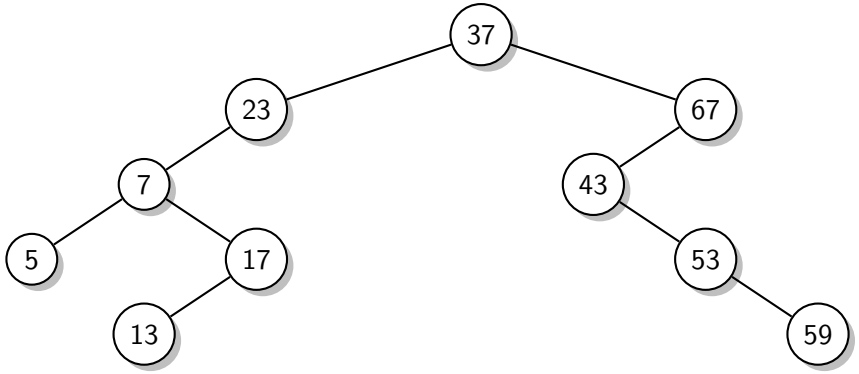
Binární vyhledávací strom – vložení klíče 5



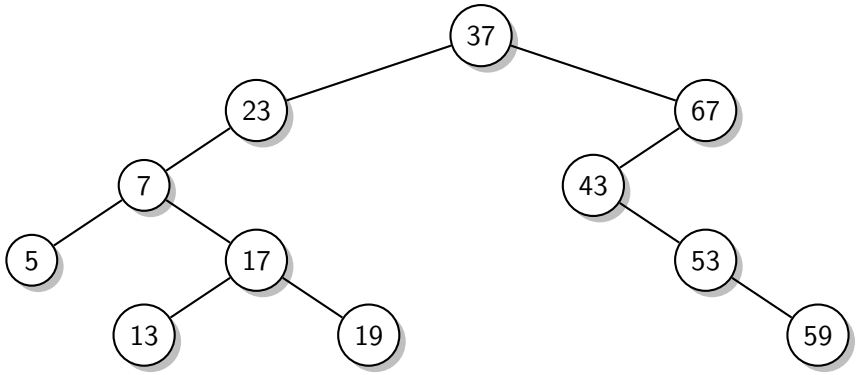
Binární vyhledávací strom – vložení klíče 17



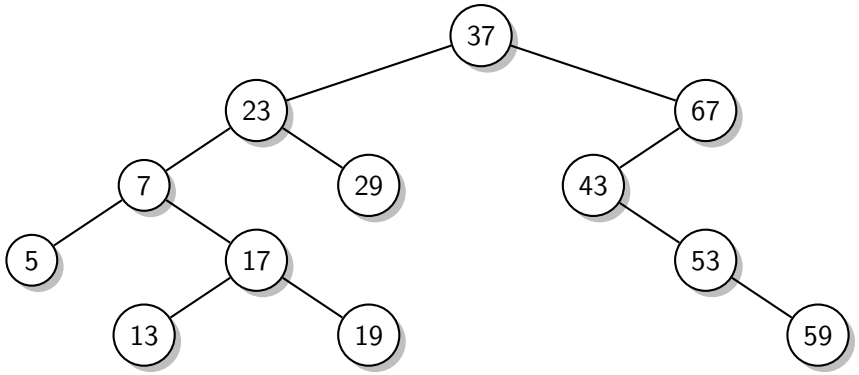
Binární vyhledávací strom – vložení klíče 13



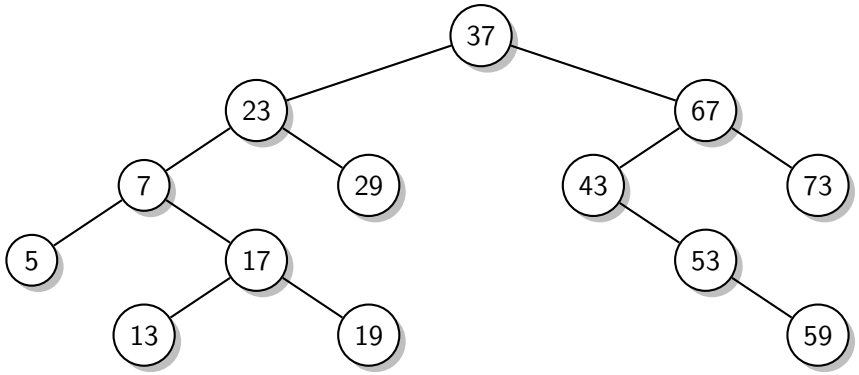
Binární vyhledávací strom – vložení klíče 19



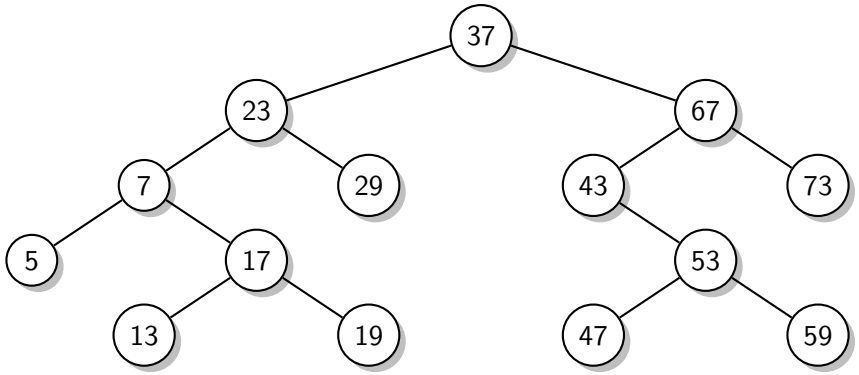
Binární vyhledávací strom – vložení klíče 29



Binární vyhledávací strom – vložení klíče 73



Binární vyhledávací strom – vložení klíče 47



Kontrolní otázky

1. Jak zní rekurzivní definice binárního stromu?
2. Je možné levého a pravého potomka daného uzlu v binárním stromu zaměňovat?
3. Co je to úplný binární strom?
4. Předpokládejme, že máme dán binární strom. Co musí platit pro klíče ve všech uzlech binárního stromu, abychom jej mohli považovat za binární **vyhledávací** strom?
5. Popište rekurzivní algoritmus vyhledání klíče x v binárním vyhledávacím stromu.
6. Z jaké vlastnosti (obecně) stromu plyne fakt, že jsme schopni v binárním vyhledávacím stromu nalézt všechny klíče, které do něj byly vloženy? Jinak řečeno, že lze navštívit všechny uzly ve stromu.

Kontrolní otázky (pokrač.)

7. Z jaké vlastnosti (obecně) stromu plyne fakt, že si nemusíme značkovat již navštívené uzly stromu? Například u průchodu grafem jsem si museli uzly označovat.
8. Popište nerekurzivní variantu vyhledání klíče x v binárním vyhledávacím stromu.
9. Popište algoritmus jak lze v binárním vyhledávacím stromu najít minimum mezi všemi v něm uloženými klíči.
10. Popište algoritmus jak lze v binárním vyhledávacím stromu najít maximum mezi všemi v něm uloženými klíči.
11. Popište algoritmus vložení klíče x do binárního vyhledávacího stromu.
12. Jakým způsobem souvisí algoritmus pro vkládání a vyhledávání v binárním vyhledávacím stromu?
13. Jaká je nejhorší podoba binárního vyhledávacího stromu? V jakou jinou datovou strukturu strom v tomto případě zdegeneruje?

Kontrolní otázky (pokrač.)

14. Jak vypadá posloupnost klíčů, jejímž vložením vznikne degenerovaný binární vyhledávací strom?
15. Lze se nějak bránit vzniku degenerovaného binárního vyhledávacího stromu?
16. Máme dáno n různých klíčů. Vložením těchto klíčů do binárního vyhledávacího stromu vznikne degenerovaný binární vyhledávací strom. Jaká je výška takto vzniklého degenerovaného binárního vyhledávacího stromu?
17. Máme dáno n různých klíčů. Tyto klíče vložíme do binárního vyhledávacího stromu, který není degenerovaný. Jaká je, v nejlepším případě, výška takto vzniklého binárního vyhledávacího stromu?
18. Předpokládejme, že máme binární vyhledávací strom s n uzly, který není degenerovaný, a hledáme klíč k. Kolik uzlů musíme v průměru projít, abychom zjistili, zda klíč k v binárním vyhledávacím stromu je nebo není?

Kontrolní otázky (pokrač.)

19. Předpokládejme, že máme binární vyhledávací strom s n uzly, který není degenerovaný, a vkládáme klíč k. Kolik uzlů musíme v průměru projít, než můžeme klíč k do stromu vložit?
20. Popište rekurzivní algoritmus, který zjistí počet uzlů ve stromu. Nápověda: zamyslete se, kolik mají uzlů stromy zmiňované v bodech 1 a 2 v definici binárního stromu.

Děkuji za pozornost

Binární vyhledávací stromy II

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Kontrolní otázky

1. Máme za úkol odebrat klíč ze stromu, čili zrušit odpovídající uzel v binárním vyhledávacím stromu. Jak postupujeme, jestliže rušený uzel nemá žádné potomky.
2. Máme za úkol odebrat klíč ze stromu, čili zrušit odpovídající uzel v binárním vyhledávacím stromu. Jak postupujeme, jestliže rušený uzel má jednoho potomka.
3. V binárním vyhledávacím stromu je dán klíč k. Jakým způsobem najdeme, či kde se v tomto stromu nachází nejbližší menší klíč než k?
4. V binárním vyhledávacím stromu je dán klíč k. Jakým způsobem najdeme, či kde se v tomto stromu nachází nejbližší větší klíč než k?
5. Máme za úkol odebrat klíč ze stromu, čili zrušit odpovídající uzel v binárním vyhledávacím stromu. Jak postupujeme, jestliže rušený uzel má dva potomky.
6. Co je to průchod binárním stromem?

Kontrolní otázky (pokrač.)

7. Lze na binární strom aplikovat obecné algoritmy průchodu grafem? Proč?
8. Jakou datovou strukturu byste použili pro průchod stromem po „patrech“?
9. Popište přímý průchod binárním stromem (anglicky preorder).
10. Popište vnitřní průchod binárním stromem (anglicky inorder).
11. Popište zpětný průchod binárním stromem (anglicky postorder).
12. Uveďte příklady využití jednotlivých způsobů průchodu binárním stromem.
13. Uvažujme libovolný způsob průchodu stromem (preorder, inorder, postorder). Z jaké vlastnosti binárního stromu plyne fakt, že si nemusíme uzly ve stromu značkovat (černá, šedá a bílá barva), abychom věděli, který uzel jsme již navštívili a přitom algoritmus skončí, tj. neuvázne v nekonečném cyklu?

Kontrolní otázky (pokrač.)

14. Uvažujme libovolný způsob průchodu stromem (preorder, inorder, postorder). Z jaké vlastnosti binárního stromu plyne fakt, že si nemusíme uzly ve stromu značkovat (černá, šedá a bílá barva), abychom věděli, který uzel jsme již navštívili a přitom žádný uzel nevynecháme?
15. Popište implementaci iterátoru v binárním vyhledávacím stromu. Jak se musíme vypořádat s faktem, že na každý uzel ve stromu mohou navazovat další dva potomci? Jakou musíme použít pomocnou datovou strukturu?
16. Jak najdeme v iterátoru další prvek pro iteraci? Předpokládejme, že iterujeme od nejmenšího prvku směrem k největšímu.
17. Jak poznáme v iterátoru konec iterace?

Kontrolní otázky (pokrač.)

18. Výsledkem algoritmu průchodu grafu do hloubky (DFS) je kostra grafu. DFS algoritmus aplikujeme na binární vyhledávací strom. Jak bude vypadat kostra grafu pro tento druh grafu? Nápověda: Kostra grafu bude shodná s binárním stromem? Nebo bude nějaký uzel či hrana chybět?
19. Mohou existovat při DFS průchodu binárním vyhledávacím stromem zpětné hrany? Nebo jen dopředné? Vysvětlete.

Děkuji za pozornost

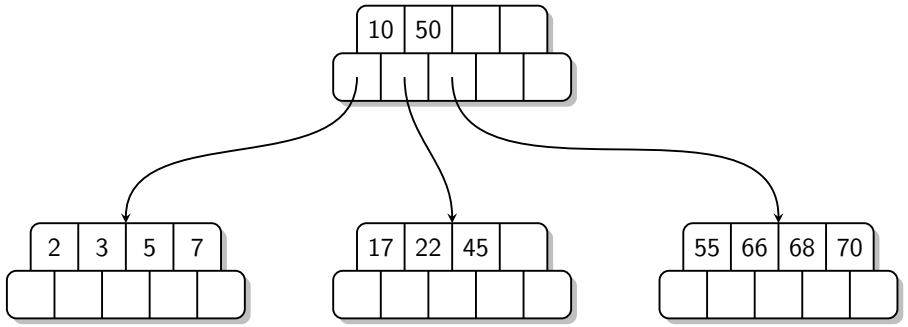
Vyvážené a vícecestné stromy

doc. Mgr. Jiří Dvorský, Ph.D.

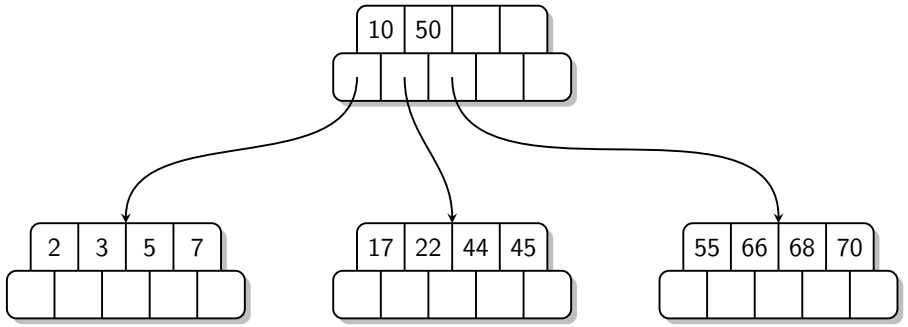
Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



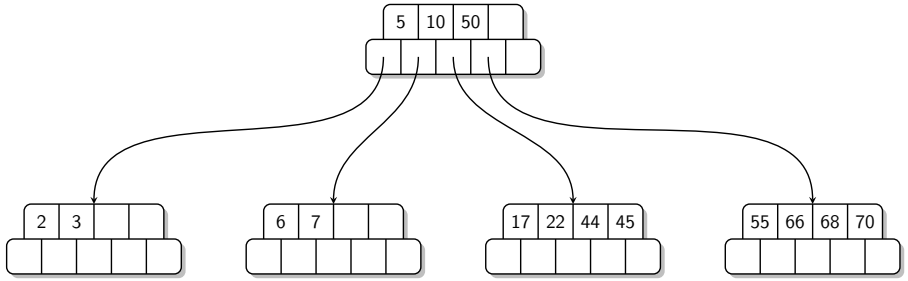
Výchozí B-strom



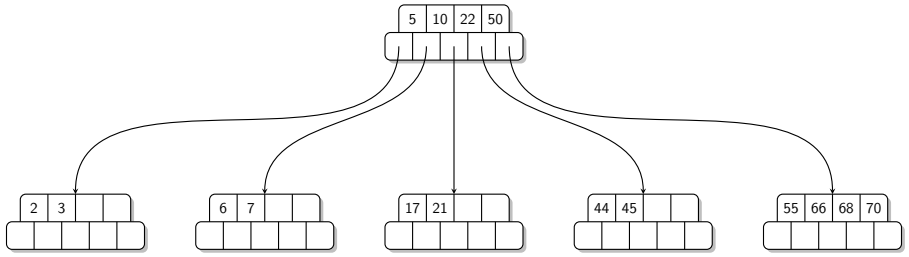
Vložení klíče 44



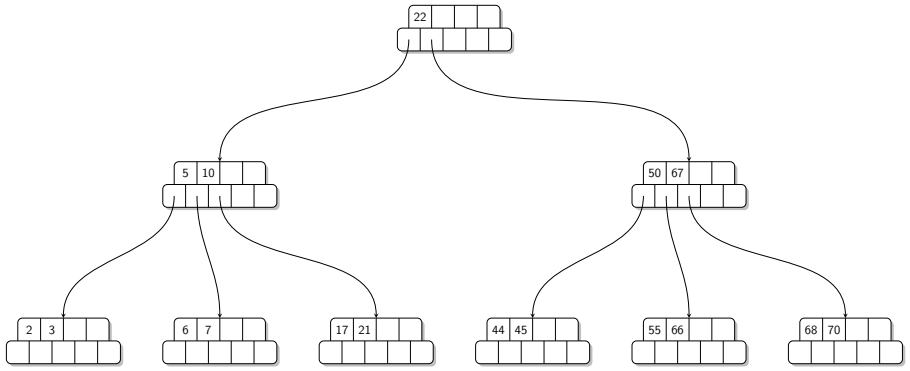
Vložení klíče 6



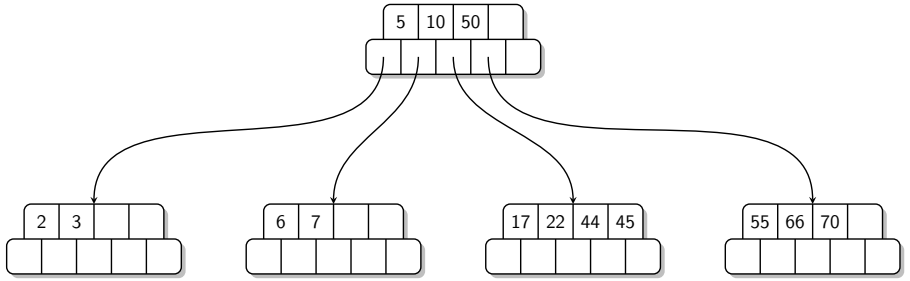
Vložení klíče 21



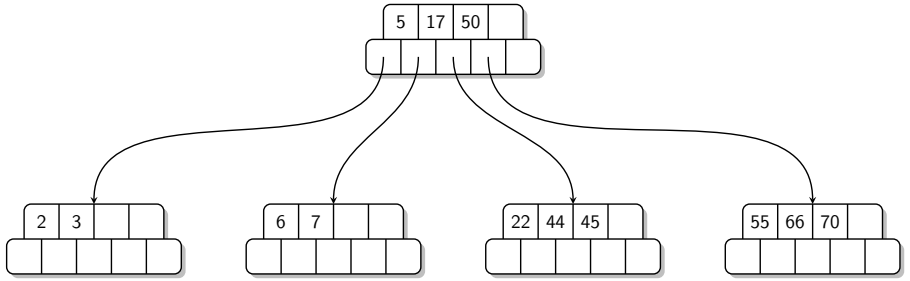
Vložení klíče 67



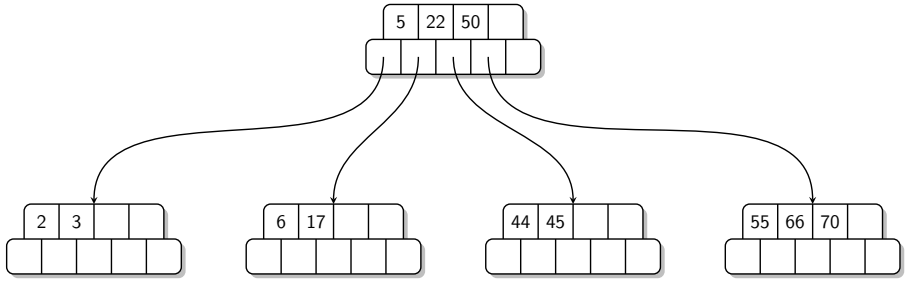
Odebrání klíče 68



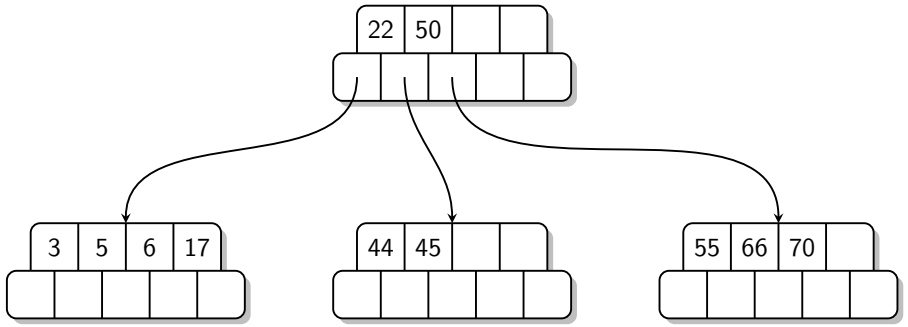
Odebrání klíče 10



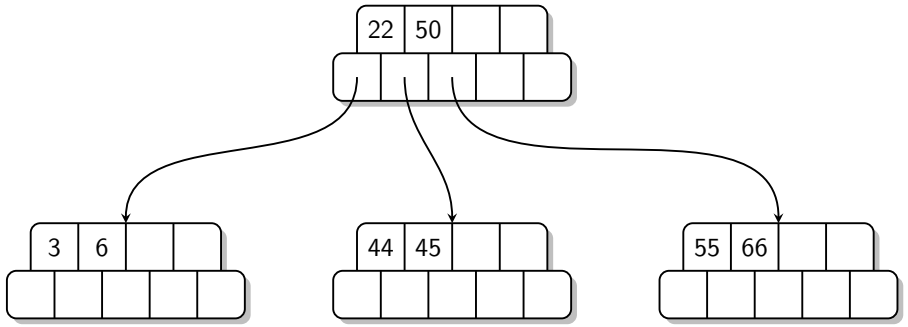
Odebrání klíče 7



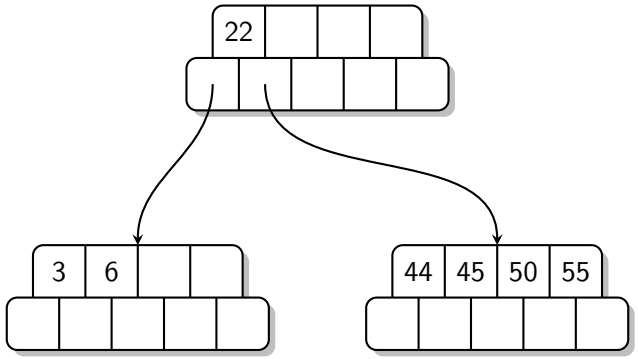
Odebrání klíče 2



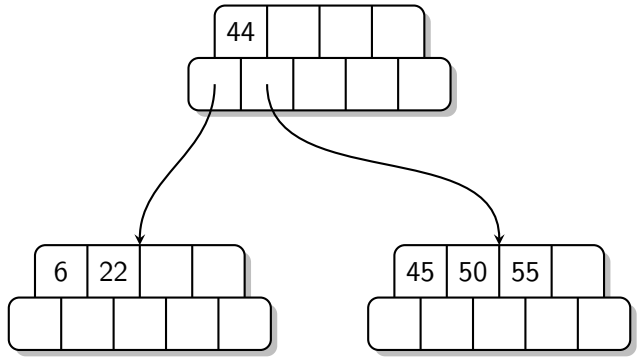
Odebrání klíčů 5, 17 a 70



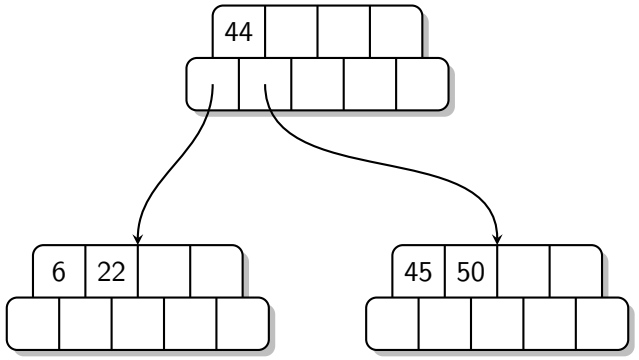
Odebrání klíče 66



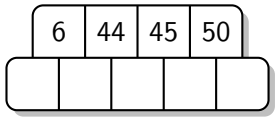
Odebrání klíče 3



Odebrání klíče 55



Odebrání klíče 22



Kontrolní otázky

1. Jak je definován dokonale vyvážený strom?
2. Jak lze snadno, pro pevně danou množinu klíčů, zkonstruovat dokonale vyvážený strom?
3. Máme danou množinu klíčů. Pro tuto množinu sestrojíme odpovídající dokonale vyvážený strom, AVL strom, binární vyhledávací strom (základní nevyvážená varianta). Který z těchto stromů bude mít minimální výšku. Poznámka: v jistých případech se může výška všech tři stromů shodovat, ale vždy tím způsobem, že výška dvou stromů se shodou okolností bude rovnat výšce třetího.
4. Jak je definováno kritérium vyváženosti AVL stromu?
5. Kdy nazýváme binární vyhledávací strom červeno-černým stromem? Co musí vše takový strom splňovat (jaká je jeho definice)?
6. Dokážete formulovat kritérium vyváženosti červeno-černého stromu?

Kontrolní otázky (pokrač.)

7. Jak se nazývají operace, které obnovují vyváženosti stromu jeho přetáčením tj. změnami vztahu rodič-potomek?
8. 2-3-4 strom obsahuje tři různé typy uzlů. Kolik obsahují jednotlivé typy uzlů klíčů?
9. 2-3-4 strom obsahuje tři různé typy uzlů. Kolik obsahují jednotlivé typy uzlů ukazatelů na potomky?
10. Co se stane s 4-uzlem jestliže je již zcela zaplněn a musíme do něj vložit další klíč?
11. Předpokládejme, že máme dán B-strom řádu n. Kolik každá stránka obsahuje nejvýše klíčů?
12. Předpokládejme, že máme dán B-strom řádu n. Kolik musí každá stránka obsahovat nejméně klíčů?
13. Předpokládejme, že máme dán B-strom řádu n. Existuje výjimka z minimálního a maximálního počtu klíčů ve stránce?

Kontrolní otázky (pokrač.)

14. Předpokládejme, že máme dán B-strom řádu n. Jestliže stránka obsahuje m klíčů, kolik má tato stránky potomků?
15. Předpokládejme, že máme dán B-strom řádu n. Jakou hloubku musí mít všechny listy ve stromu?
16. Předpokládejme, že máme dán B-strom řádu n. Dále předpokládejme, že klíče ve stránce ukládáme v poli a klíče v tomto pole udržujeme setříděné. Jaký byste použili efektivní algoritmus pro nalezení klíče k v tomto poli?
17. Předpokládejme, že máme dán B-strom řádu n. Dále máme stránku S (S není kořen stromu), která je zcela zaplněna, tj. obsahuje $2n$ klíčů. Do této stránky chceme přidat další klíč. Popište co se s touto stránkou stane a co se stane s jednotlivými klíči (skupinami klíčů).

Kontrolní otázky (pokrač.)

18. Předpokládejme, že máme dán B-strom řádu n. A dále máme kořenovou stránku B-stromu, která je zcela zaplněna, tj. obsahuje $2n$ klíčů. Do této stránky chceme přidat další klíč. Popište co se s touto stránkou stane a co se stane s jednotlivými klíči (skupinami klíčů).
19. Předpokládejme, že máme dán B-strom řádu n. Ve stromu je celkem M klíčů. Jaká bude přibližně výška tohoto B-stromu?
20. Je B-strom vyvážený nebo ne?
21. Co je to faktor využití paměti u B-stromu?
22. Předpokládejme, že máme dán B-strom řádu n. Máme stránku, které obsahuje více než n klíčů, jeden z těchto klíčů chceme ze stránky odebrat. Jak to provedete?
23. Předpokládejme, že máme dán B-strom řádu n. Máme stránku, které obsahuje přesně n klíčů, jeden z těchto klíčů chceme ze stránky odebrat. Jak to provedete?

Kontrolní otázky (pokrač.)

24. Může B-strom snížit svoji výšku? Kdy k tomu dojde?
25. Může B-strom zvýšit svoji výšku? Kdy k tomu dojde?

Děkuji za pozornost

Hašování

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Kontrolní otázky

1. Co je to univerzum klíčů?
2. Jaký bývá typický poměr velikostí mezi univerzem klíčů a množinou skutečně zpracovávaných klíčů, tj. těmi, které jsou skutečně uloženy v tabulce?
3. Co je to kolize v hašovací tabulce?
4. Lze kolizím zcela zabránit?
5. Jakým principem řeší separátní řetězení kolize?
6. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. Popište, jak se do tabulky vloží klíč k.
7. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. Popište, jak se v tabulce nalezne klíč k.

Kontrolní otázky (pokrač.)

8. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. Popište, jak se z tabulky odebere klíč k. Je to vůbec možné?
9. Co je to faktor naplnění α? Jak se vypočítá?
10. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny separátním řetězením. V tabulce je uloženo celkem n klíčů. Kolik prvků v tabulce musíme prozkoumat, abychom rozhodli, zda se tam prvek nalézá nebo ne. Jak tento počet souvisí s faktorem naplnění α?
11. Může být u hašovacích tabulek s kolizemi řešenými pomocí separátního řetězení faktor naplnění α větší než jedna, tj. $\alpha > 1$?
12. Může být u hašovací tabulky, kde jsou kolize řešeny otevřeným adresováním faktor naplnění α větší než jedna, tj. $\alpha > 1$?
13. Co je to jednoduché uniformní hašování?

Kontrolní otázky (pokrač.)

14. Co je to uniformní hašování?
15. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny otevřeným adresováním, metodou lineárních pokusů. Popište, jak se do tabulky vloží klíč k.
16. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny otevřeným adresováním, metodou lineárních pokusů. Popište, jak se v tabulce nalezne klíč k.
17. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny otevřeným adresováním, metodou lineárních pokusů. Popište, jak se z tabulky odebere klíč k. Je to vůbec možné?
18. Popište princip řešení kolizí pomocí lineárních pokusů u hašovací tabulky s otevřeným adresováním.
19. Popište princip řešení kolizí pomocí kvadratických pokusů u hašovací tabulky s otevřeným adresováním.

Kontrolní otázky (pokrač.)

20. Popište princip řešení kolizí pomocí dvojitého hašování u hašovací tabulky s otevřeným adresováním.
21. Co je to primární shlukování?
22. Co je to sekundární shlukování?
23. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty, kolize jsou řešeny některou z metod otevřeného adresování. Kolik pokusů můžu maximálně provést při hledání klíče k?
24. Předpokládejme, že máme dánu hašovací tabulku s celkem m sloty. Kolize jsou řešeny separátním řetězením. Dále předpokládejme, že hašovací hodnota klíče k je $h(k)$. Který slot (sloty) musím při hledání klíče k v tabulce prohledat:
    - 24.1 všechny,
    - 24.2 jen s indexem $h(k)$,
    - 24.3 slot s indexem $h(k)$ a sousední sloty s indexy $h(k) - 1$ a $h(k) + 1$.
25. Jaká čísla jsou vhodnými kandidáty na velikost hašovací tabulky?

Děkuji za pozornost

Vyhledávání v textu

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



Definice základních pojmů

Definice

Abeceda Σ je konečná neprázdná množina symbolů.

Příklady:

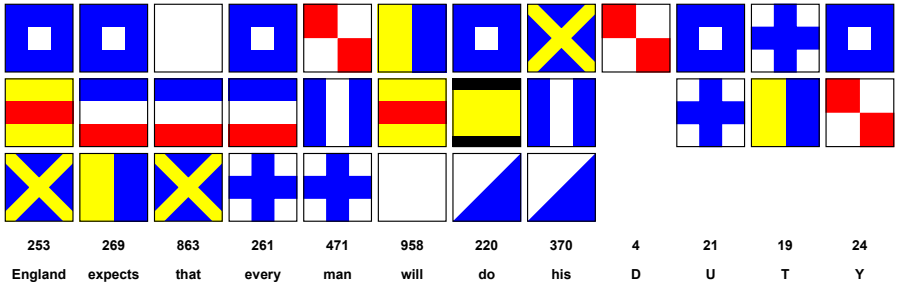
- binární abeceda $\Sigma = \{0, 1\}$,
- Morseova abeceda $\Sigma = \{., -, /\}$,
- abeceda aminokyselin DNA $\Sigma = \{A, C, G, T\}$,
- anglická abeceda $\Sigma = \{a, \ldots, z\}$.

Vlajková abeceda

- abeceda může mít i fyzickou podobu, viz například námořní vlajková abeceda, `en.wikipedia.org/wiki/International_maritime_signal_flags`
- současná podoba z roku 1931,
- celkem 40 vlajek – 26 písmen, 10 číslic, 1 potvrzovací vlajka a 3 opakovací,
- každá vlajka reprezentuje i jednu často opakovanou větu, například vlajka O znamená „Muž přes palubu“,
- případně lze vlajky kombinovat do číselných kódů odkazujících do smluveného slovníku.

Vlajková abeceda – příklad

Slavný signál admirála Nelsona „England expects that every man will do his duty“ vyslaný v roce 1805 před bitvou u Trafalgaru vypadá ve vlajkové abecedě takto:



`en.wikipedia.org/wiki/England_expects_that_every_man_will_do_his_duty`

Definice základních pojmů

Definice

Konečná posloupnost x symbolů ze Σ se nazývá řetězec nad Σ.

Definice

Prázdná posloupnost se nazývá prázdný řetězec, značíme ji ε.

Definice

Délka řetězce x se značí $|x|$ a rovná se počtu výskytů symbolů v něm obsažených, $|\varepsilon| = 0$.

Definice

Množinu všech řetězců nad abecedou Σ bez prázdného řetězce budeme značit Σ^+. Množinu všech řetězců nad abecedou Σ, včetně prázdného řetězce, budeme značit Σ^*.

Definice základních pojmů (pokrač.)

Věta

Je zřejmé, že

$$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$$

Definice

Nechť x a y jsou libovolné řetězce, pak xy označuje konkatenaci (zřetězení) řetězců x a y.

Věta

Konkatenace dvou řetězců není obecně komutativní, tj.

$$\exists x, y \in \Sigma^+ : |xy| \neq |yx|$$

Dále je zřejmé, že

$$|xy| = |yx| = |x| + |y|$$

Definice základních pojmů (pokrač.)

Definice

Řetězec u se nazývá předponou (prefixem) řetězce w, jestliže existuje řetězec v (i prázdný) takový, že $w = uv$.

Definice

Řetězec v se nazývá příponou (sufixem) řetězce w, jestliže existuje řetězec u (i prázdný) takový, že $w = uv$.

Definice

Řetězec y se nazývá podřetězcem (faktorem) řetězce w, jestliže existují řetězce u a v (i prázdné) takové, že $w = uzv$.

Definice základních pojmů – příklady

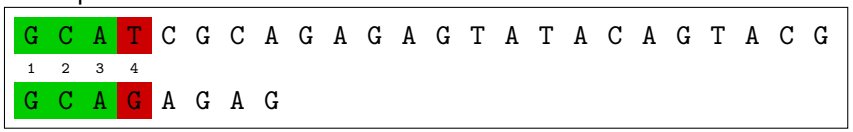
- Abeceda $\Sigma = \{A, C, G, T\}$, řetězce $x = GCAGAGAG$ a $y = TAA$, $|x| = 8$ a $|y| = 3$.
- Konkatenace $xy = GCAGAGAGTAA$, $yx = TAAGCAGAGAG$, $|xy| = |yx| = |x| + |y| = 11$.
- Prefixem řetězce y jsou řetězce T, TA, TAA.
- Sufixem řetězce y jsou řetězce A, AA, TAA.
- Podřetězcem řetězce y jsou řetězce A, T, TA, AA, TAA.

Brute force algoritmus

```
1 size_t BruteForce(const string& Text, const string& Pattern, const
      size_t StartPosition)
2 {
3    for (size_t i = StartPosition; i < Text.length() - Pattern.length();
         i++)
4    {
5       size_t j = 0;
6       while (j < Pattern.length())
7       {
8          if (Text[i + j] != Pattern[j])
9             break;
10         j += 1;
11      }
12      if (j == Pattern.length())
13         return i;
14   }
15   return PatternNotFound;
16 }
```

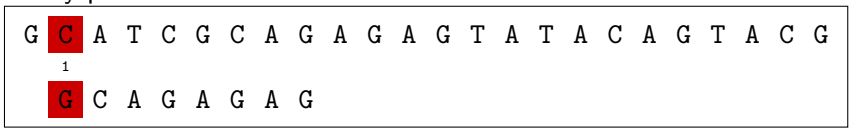

Brute force algoritmus

První pokus



Posun o 1 znak

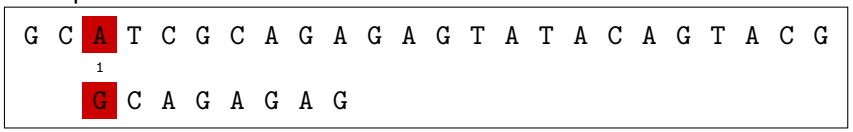
Druhý pokus



Posun o 1 znak

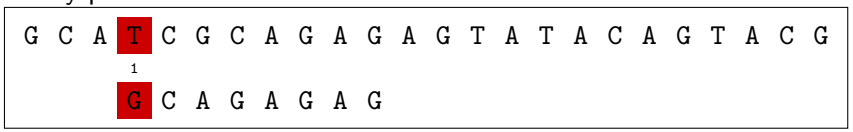
Brute force algoritmus (pokrač.)

Třetí pokus



Posun o 1 znak

Čtvrtý pokus



Posun o 1 znak

Brute force algoritmus (pokrač.)

Pátý pokus



Posun o 1 znak

Šestý pokus



Posun o 1 znak

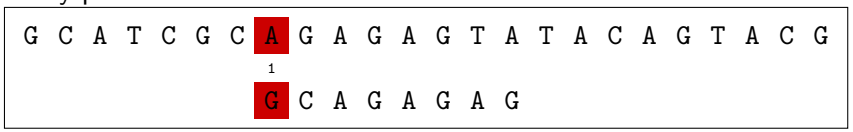
Brute force algoritmus (pokrač.)

Sedmý pokus



Posun o 1 znak

Osmý pokus



Posun o 1 znak

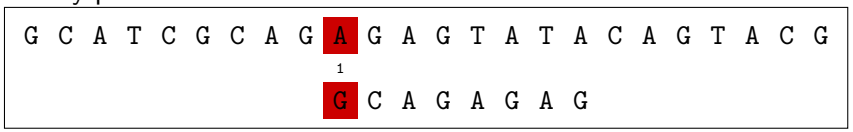
Brute force algoritmus (pokrač.)

Devátý pokus



Posun o 1 znak

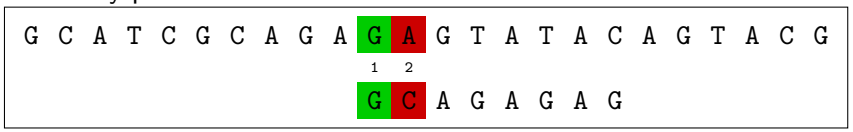
Desátý pokus



Posun o 1 znak

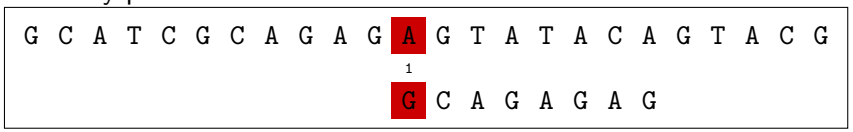
Brute force algoritmus (pokrač.)

Jedenáctý pokus



Posun o 1 znak

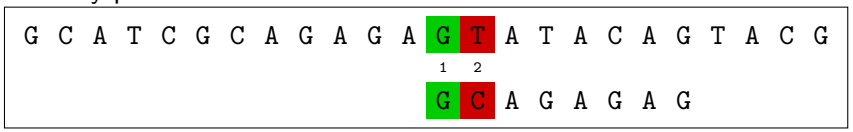
Dvanáctý pokus



Posun o 1 znak

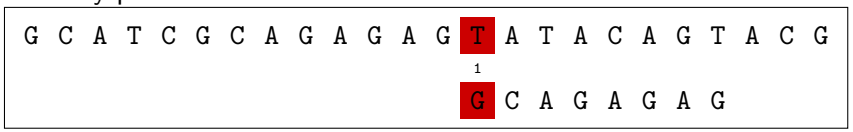
Brute force algoritmus (pokrač.)

Třináctý pokus



Posun o 1 znak

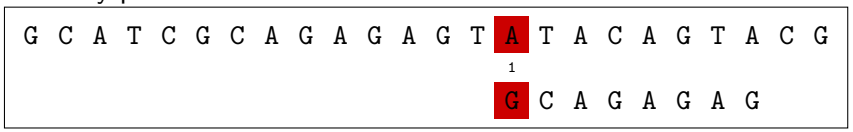
Čtrnáctý pokus



Posun o 1 znak

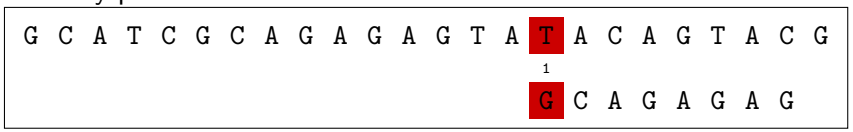
Brute force algoritmus (pokrač.)

Patnáctý pokus



Posun o 1 znak

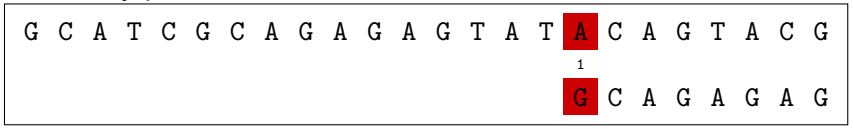
Šesnáctý pokus



Posun o 1 znak

Brute force algoritmus (pokrač.)

Sedmnáctý pokus

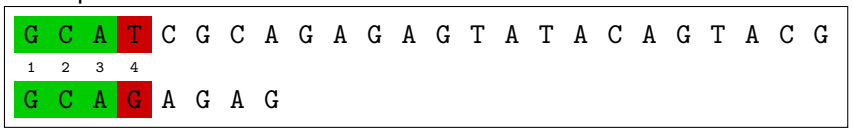


Posun o 1 znak

Algoritmus provedl celkem 30 porovnání znaků.

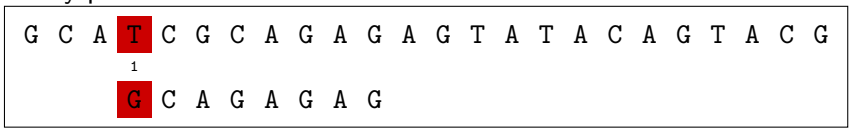
Morris-Prattův algoritmus

První pokus



Posun o $i - Next[i] = 3 - 0 = 3$ znaky

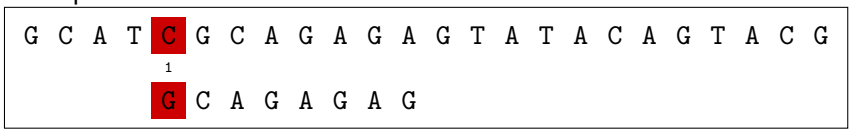
Druhý pokus



Posun o $i - Next[i] = 0 - (-1) = 1$znak

Morris-Prattův algoritmus (pokrač.)

Třetí pokus



Posun o $i - Next[i] = 0 - (-1) = 1$ znak

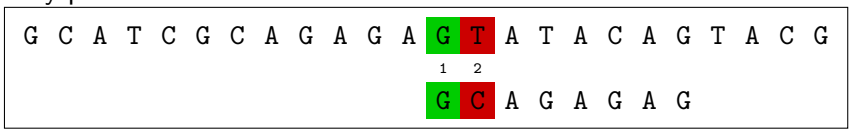
Čtvrtý pokus



Posun o $i - Next[i] = 8 - 1 = 7$ znaků

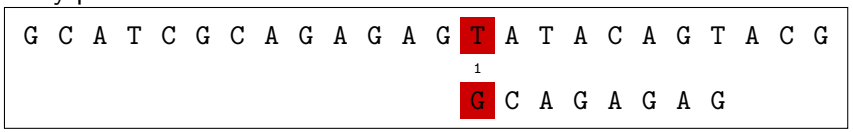
Morris-Prattův algoritmus (pokrač.)

Pátý pokus



Posun o $i - Next[i] = 1 - 0 = 1$ znak

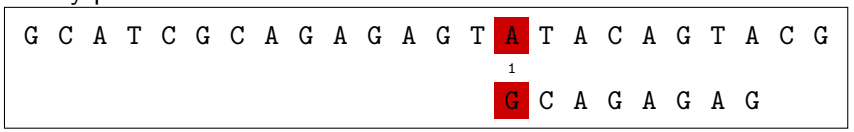
Šestý pokus



Posun o $i - Next[i] = 0 - (-1) = 1$ znak

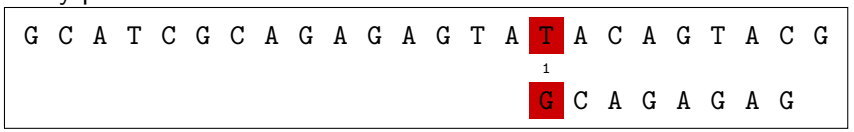
Morris-Prattův algoritmus (pokrač.)

Sedmý pokus



Posun o $i - Next[i] = 0 - (-1) = 1$ znak

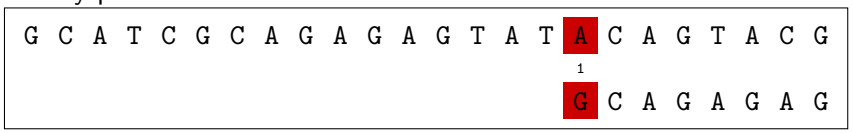
Osmý pokus



Posun o $i - Next[i] = 0 - (-1) = 1$ znak

Morris-Prattův algoritmus (pokrač.)

Devátý pokus

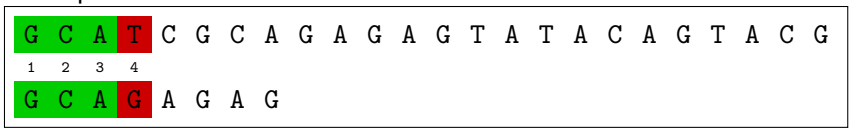


Posun o $i - Next[i] = 0 - (-1) = 1$ znak

Algoritmus provedl celkem 19 porovnání znaků.

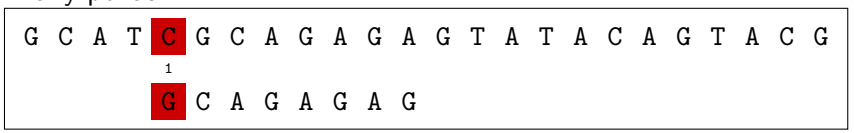
Knuth-Morris-Prattův algoritmus

První pokus



Posun o 4 znaky, tj. $i - Next[i] = 3 - (-1)$

Druhý pokus



Posun o 1 znak, tj. $i - Next[i] = 0 - (-1)$

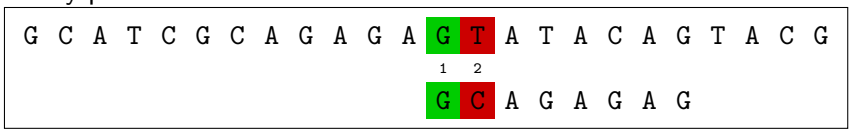
Knuth-Morris-Prattův algoritmus (pokrač.)

Třetí pokus



Posun o 7 znaků, tj. $i - Next[i] = 8 - 1$

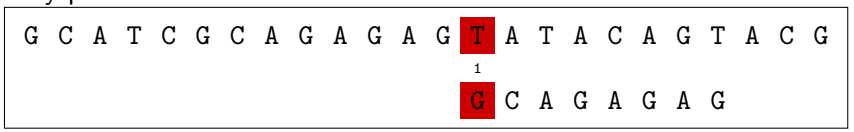
Čtvrtý pokus



Posun o 1 znak, tj. $i - Next[i] = 1 - 0$

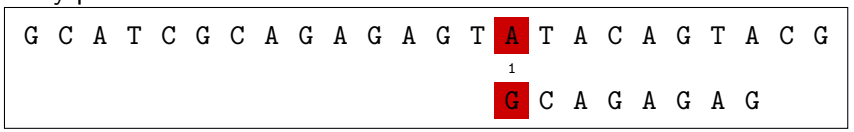
Knuth-Morris-Prattův algoritmus (pokrač.)

Pátý pokus



Posun o 1 znak, tj. $i - Next[i] = 0 - (-1)$

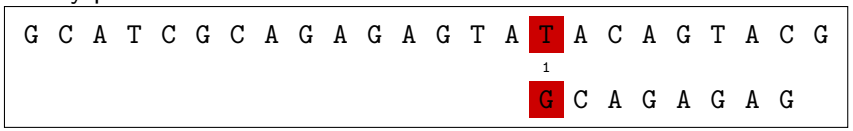
Šestý pokus



Posun o 1 znak, tj. $i - Next[i] = 0 - (-1)$

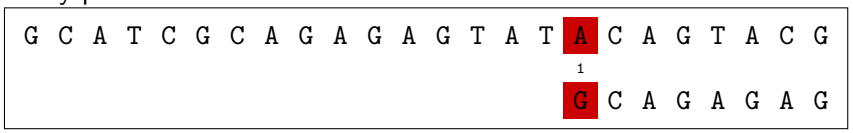
Knuth-Morris-Prattův algoritmus (pokrač.)

Sedmý pokus



Posun o 1 znak, tj. $i - Next[i] = 0 - (-1)$

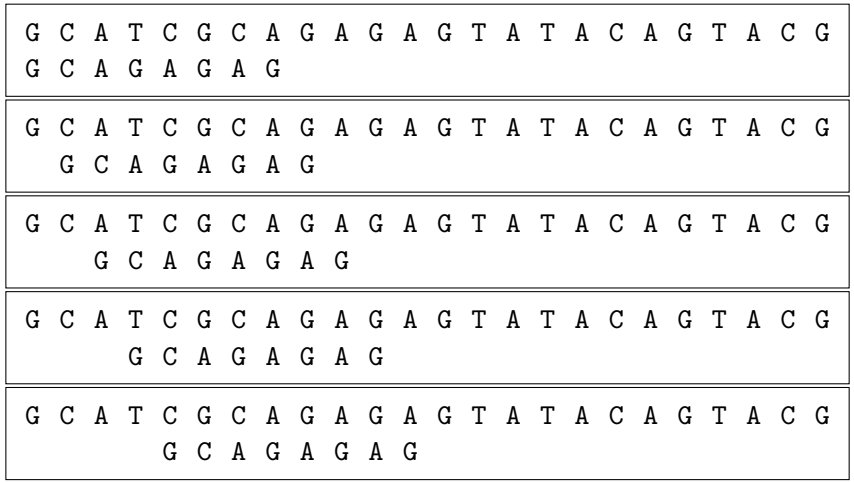
Osmý pokus



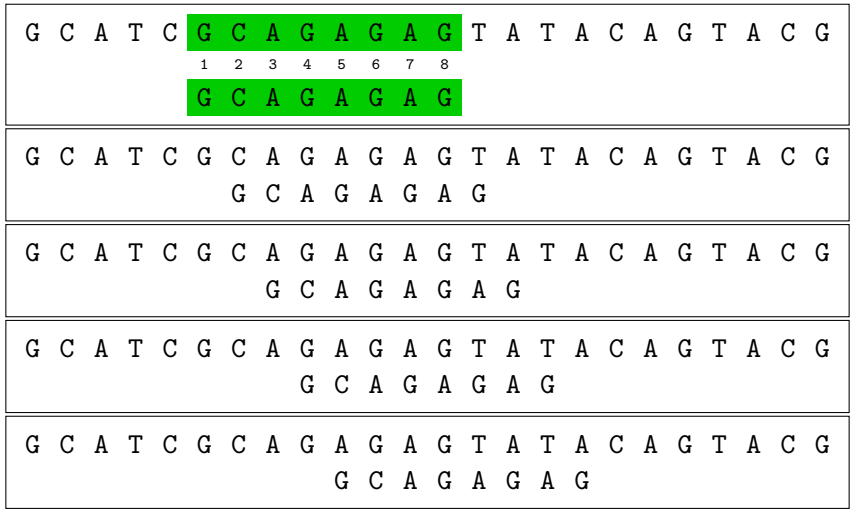
Posun o 1 znak, tj. $i - Next[i] = 0 - (-1)$

Algoritmus provedl celkem 18 porovnání znaků.

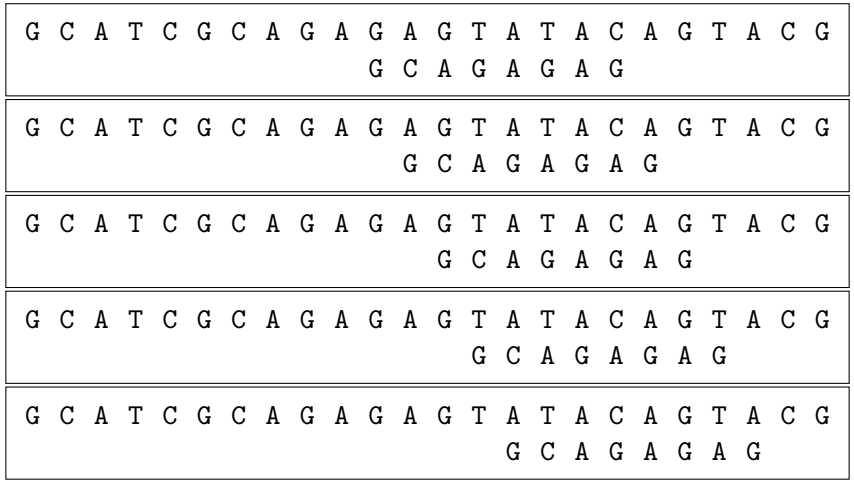
KarpRabin



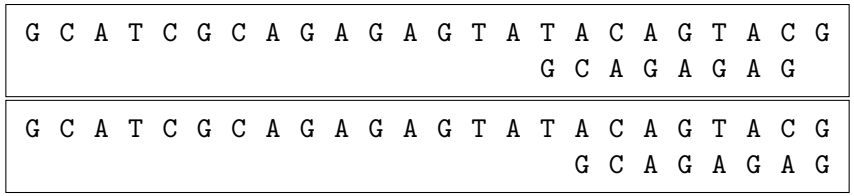
KarpRabin (pokrač.)



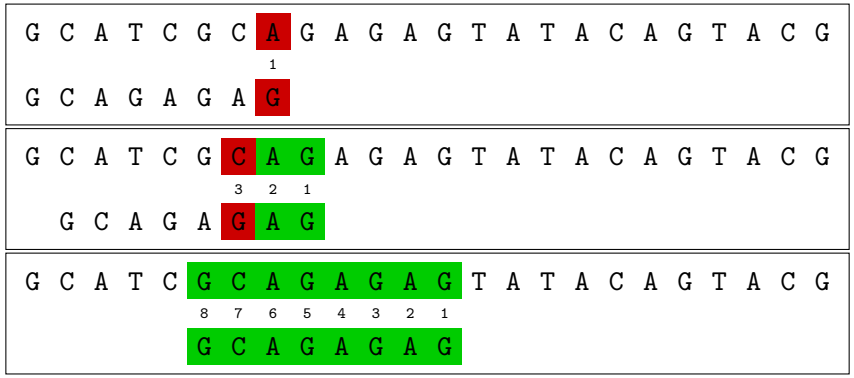
KarpRabin (pokrač.)



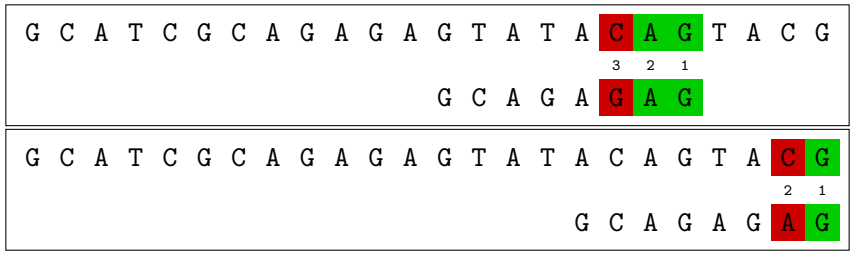
KarpRabin (pokrač.)



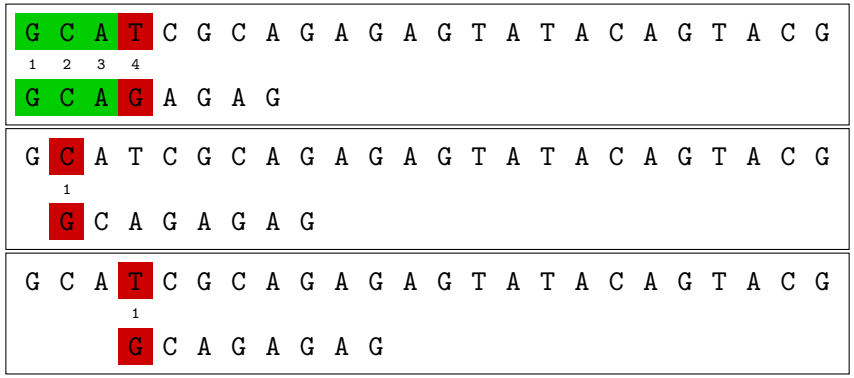
BoyerMoore



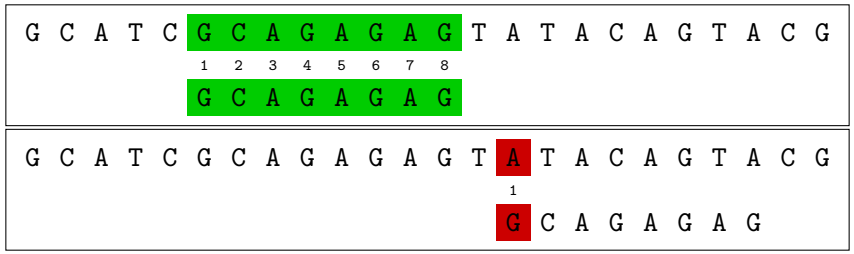
BoyerMoore (pokrač.)



QuickSearch



QuickSearch (pokrač.)



Kontrolní otázky

1. Definujte problém vyhledání vzorku v textu. Kde se lze s těmito problémy setkat?
2. Co je to předzpracování textu a vzorku?
3. Uveďte algoritmus, který pro svůj běh nepotřebuje předzpracovat ani text ani vzorek.
4. Uveďte příklad algoritmu (algoritmů), který předzpracovává vzorek ale nepředzpracovává prohledávaný text.
5. Uveďte příklad vyhledávací metody, která nepředzpracovává vzorek ale předzpracovává prohledávaný text.
6. Uveďte příklad vyhledávací metody, která předzpracovává vzorek i prohledávaný text.
7. Co je to abeceda? Označme ji například Σ.
8. Co je to řetězec nad abecedou Σ? Jak je definována délka řetězce?

Kontrolní otázky (pokrač.)

9. Máte dánu abecedu Σ a přirozená čísla m a n. Jak budou vypadat řetězce a^m a $a^m b^n$, kde $a, b \in \Sigma$?
10. Co je to prázdný řetězec? Jak jej značíme? Jaká je jeho délka?
11. Máte dánu abecedu Σ. Co označují množiny Σ^+ a Σ^*?
12. Co je to předpona (prefix) řetězce w?
13. Co je to přípona (sufix) řetězce w?
14. Co je to podřetězec (faktor) řetězce w?
15. Co je to hranice řetězce w?
16. Popište princip elementárního algoritmu (BruteForce Algorithm).
17. Jaká je časová složitost elementárního algoritmu? Jaké operace nás zajímají u výpočtu složitosti vyhledávacích algoritmů?
18. Jaká je očekávaná časová složitost elementárního algoritmu pro text a vzorek z přirozených jazyků?

Kontrolní otázky (pokrač.)

19. Elementární algoritmus provádí jednu činnost se vzorkem velice neefektivně. Tato neefektivita mu znemožňuje dosahovat menší časové složitosti při vyhledávání. O jako činnost je vzorkem jde a jak tuto neefektivitu řeší ostatní algoritmy? Například Knuth-Morris-Prattův?
20. Popište princip Shift-Or algoritmu. Jaké dvě bitové operace se tu používají?
21. Jaká je časová složitost Shift-Or algoritmu?
22. Popište princip Karp-Rabinova algoritmu. K čemu slouží u tohoto algoritmu hašovací funkce (Pozn. Vzpomeňte si na kapitolu o hašování a pohled na hašovací funkci jako na „negativní filtr“.)?
23. Co je to sousměrné a protisměrné vyhledávání?
24. Popište princip Boyer-Mooreova algoritmu. Soustřeďte se na využití protisměrného vyhledávání pro efektivní detekci (ne)výskytu vzorku v textu.

Kontrolní otázky (pokrač.)

25. Jaká je složitost Boyer-Mooreova algoritmu v nejhorším případě a jaká v nejlepším? Pro jakou kombinaci vzorku a textu dojde k nejhoršímu případu?

Děkuji za pozornost