KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

ZÁKLADNÍ ALGORITMY

ARNOŠT VEČERKA

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2007

Abstrakt

Tento text distančního vzdělávání seznamuje se základními algoritmy, které se používají jak samostatně, tak i jako součást jiných, složitějších algoritmů nebo úloh. Na začátku stručný úvod do časové složitosti algoritmů. Dále je na začátku je stručný přehled datových struktur, zejména jsou zde popsány dynamické datové struktury. Hlavní část tohoto studijního materiálu tvoří dvě třídy algoritmů. V první z nich jsou algoritmy vyhledávání, druhá obsahuje algoritmy třídění.

Cílová skupina

Text je primárně určen pro posluchače prvního bakalářského studijního programu Aplikovaná informatika na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Může však sloužit komukoli se zájmem o algoritmy a jejich použití. Text nepředpokládá žádné vstupní znalosti.

Obsah

1 Algoritmus

Algoritmus je popis určitého postupu. Příkladem algoritmu by mohl být recept na přípravu jídla obsažený v kuchařské knize. My se zde přirozeně nebudeme zabývat algoritmy vaření, ale postupy některých často používaných výpočtů. Pojem výpočet mnohdy vytváří asociaci, že jde o výpočet číselné povahy. Zde je tento pojem použit v širším slova smyslu, označuje zde nějaké zpracování údajů. Typ těchto údajů může být různý, mohou to být jak čísla, tak i textové řetězce apod.

Průvodce studiem

Algoritmy prolínají celý náš život. Ovšem běžně se jim neříká algoritmy, ale návody, pokyny k používání atd. Užitečnost a význam algoritmů oceníme zejména, když se snažíme sami přijít na to, jak se s novou věcí zachází. Často po delší době neúspěchů nakonec rezignujeme a přečteme si přiložený návod.

1.1 Složitost algoritmu

Studijní cíle: Zavést a vysvětlit pojem časové složitosti algoritmu a objasnit její důležitost pro výběr konkrétního algoritmu.

Klíčová slova: Paměťová složitost, časová složitost, polynomická složitost.

Potřebný čas: 1 hodina

Výrazným rysem algoritmů je jejich složitost. Vraťme se k našemu původnímu příkladu receptu na přípravu jídla. Bude záležet na tom, pro kolik osob jídlo budeme připravovat. Čím více těchto osob bude, tím jednak budeme potřebovat větší nádobí a dále nám typicky bude i příprava jídla trvat déle. Budeme třeba muset oškrábat více brambor atd. Stejně tak při výpočtu čím více údajů budeme zpracovávat, tím větší množství paměti budeme potřebovat pro jejich uložení a rovněž čas výpočtu bude delší. Závislost velikosti paměti potřebné pro výpočet na počtu zpracovávaných údajů nazýváme paměťovou složitostí algoritmu. Závislost doby výpočtu na počtu zpracovávaných údajů nazýváme časovou složitostí algoritmu. My se u jednotlivých algoritmů budeme zabývat jen jejich časovou složitostí. Paměťová složitost přirozeně má také svůj význam. Nicméně při současných velikostech pamětí počítačů nebývá u většiny algoritmů omezujícím faktorem pro jejich použití a není tudíž pro nás tak důležitá jako časová složitost.

Časová složitost algoritmu je významnější než jeho paměťová složitost.

Vyjadřovat časovou složitost přímo v časových jednotkách by bylo velmi obtížné a rovněž i problematické. Například vezměme zcela jednoduchou úlohu, kdy máme vypočítat součet n čísel. Zřejmě bychom postupovali tak, že bychom vzali první číslo a k němu postupně přičítali 2., 3. až n-té číslo. Celkově bychom udělali n-1 operací přičtení. Pokud bychom to dělali ručně, pak by celkový čas byl n-1 násobkem času, který potřebujeme pro přičtení jednoho čísla. Jestliže to ale budeme dělat pomocí programu, bude určení času potřebného pro přičtení jednoho čísla obtížné. Neboť program většinou píšeme v nějakém programovacím jazyce. Jak dlouho v něm bude trvat přičtení jednoho čísla, přímo nevíme. Přesněji řečeno je to značně obtížné určit, neboť program je před výpočtem nejdříve přeložen do jazyka procesoru, což znamená, že bychom museli časovou složitost počítat ze strojových instrukcí přeloženého programu. To ale bohužel není nijak jednoduché. Navíc doba výpočtu bude zřejmě také záviset na tom, jak rychlý počítač pro výpočet použijeme. Abychom tyto nepříjemné aspekty eliminovali, počítáme časovou složitost nikoliv v časových jednotkách, ale v počtech základních

Časová složitost se vyjadřuje ne v časových jednotkách, ale pomocí základních operací algoritmu.

operacích, na kterých je příslušný algoritmus založen. V našem příkladu součtu je základní operací přičtení a časová složitost tohoto algoritmu je vyjádřena lineární funkcí n-1.

Obecně je časová složitost vyjádřena matematickými funkcemi, jejichž argumentem je počet údajů, které zpracováváme. Velmi často tyto funkce jsou přímo polynomy, z ostatních funkcí se poměrně často vyskytuje logaritmická funkce. Např.

$$\frac{n^2}{4} - n + 3 \qquad 3n\log_2(n-1) \qquad \text{atd.}$$

Časová složitost nám sice neudává přímo čas výpočtu, ale dá se z ní odvodit, jak dlouho řádově bude výpočet pro náš počet údajů trvat (vteřiny, minuty, hodiny atd.). To nám stačí, abychom posoudili, zda daný algoritmus je pro nás použitelný. Dále je časová složitost důležitá v případě, kdy máme pro danou úlohu více algoritmů a máme se rozhodnout, který z nich použít. Často máme na jedné straně algoritmy, které jsou jednoduché, ale mají větší časovou složitost, a na druhé straně máme algoritmy s příznivější časovou složitosti, které jsou ale komplikovanější, čímž jejich realizace, tj. naprogramování je obtížnější. V tomto případě nám pro menší počty údajů stačí jednodušší algoritmus, a máme-li hodně údajů, musíme zvolit sice pracnější, ale na druhé straně rychlejší algoritmus.

Uvažujme nyní následující tři funkce:

$$\frac{n^2}{4} - n + 3 \qquad \frac{n^2}{4} \qquad n^2$$

Ta první funkce nechť je časovou složitostí nějakého konkrétního algoritmu. Druhou jsme dostali zanedbáním jejího lineárního a absolutního členu a v třetí jsme navíc odstranili koeficient u kvadratického členu. Všechny tři funkce jsou kvadratickým polynomem. Ta první vyjadřuje odvozenou složitost, naproti tomu ta poslední je z nich nejjednodušší. Vzhledem k tomu, že u všech jde o polynom stejného řádu, mají obdobný nárůst funkční hodnoty při rostoucím počtu údajů, tj. při zvyšující se hodnotě argumentu n. Už jsme uvedli, že časovou složitost používáme nikoliv k přesnému určení doby výpočtu, ale k jejímu řádovému odhadu. Pro tento účel nám místo první funkce, sice přesné, ale poměrně složité zcela vyhoví mnohem jednodušší poslední funkce n^2. Pro tento účel se používá operátor Θ, který vyjadřuje, že funkce v něm uvedená má stejnou míru složitosti jako funkce, jež je přesným vyjádřením časové složitosti algoritmu. Tj. místo abychom psali, že algoritmus má časovou složitost

$$\frac{n^2}{4} - n + 3 \quad ,$$

napíšeme jen, že jeho časová složitost je

$$\Theta(n^2) \ .$$

Matematicky skutečnost, že dvě funkce časové složitosti mají stejnou míru nárůstu hodnoty, můžeme charakterizovat tak, že jejich poměr (podíl) konverguje ke konečné nenulové hodnotě pro rostoucí argument n. Je-li tedy časová složitost algoritmu dána funkcí $f(n)$ a dále máme funkci $g(n)$ takovou, že je splněno

$$0 < \lim_{n\to+\infty} \frac{f(n)}{g(n)} < +\infty$$

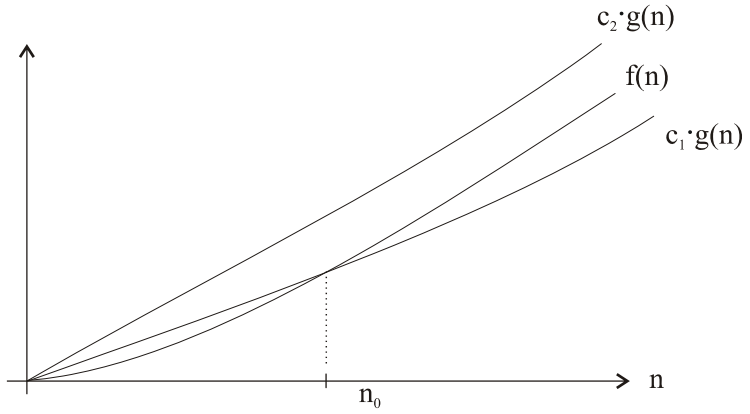
pak můžeme napsat, že časová složitost algoritmu je

$$\Theta(g(n)) \ .$$

Jiný možný způsob, jak funkci $g(n)$ definovat, je, že musí existovat dvě kladná nenulová čísla c_1 a c_2 a přirozené číslo n_0 taková, že platí

$$c_1.g(n) \leq f(n) \leq c_2.g(n) \quad \text{pro} \quad n \geq n_0 \ .$$

Uvedený vztah je znázorněn na následujícím obrázku.



Například v již uvedeném příkladu je limita podílu vlastní funkce časové složitosti algoritmu a funkce n^2:

$$\lim_{n\to+\infty} \frac{\frac{1}{4}n^2 - n + 3}{n^2} = \frac{1}{4} \ .$$

Pomocí limity můžeme také charakterizovat, kdy je časová složitost jednoho algoritmu lepší ve srovnání s druhým algoritmem. Mějme dva algoritmy a nechť první má časovou složitost vyjádřenou funkcí *f(n)* a druhý má časovou složitost *g(n)*. Jestliže bude platit

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$$

pak zřejmě funkce *f(n)* roste pomaleji než funkce *g(n)* a tudíž první algoritmus bude mít lepší časovou složitost. Naopak bude-li platit

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = +\infty$$

bude mít lepší časovou složitost druhý algoritmus.

Časovou složitost používáme k odhadu doby výpočtu.

Přejděme nyní k vlastnímu časovému odhadu. Čas potřebný pro výpočet jedné základní operace algoritmu bude záviset na tom, jak je tato operace komplikovaná. V přeloženém programu operace reprezentuje určitý počet strojových instrukcí a čím je operace komplikovanější, tím více strojových instrukcí v přeloženém programu zahrnuje a tím také delší bude doba jejího výpočtu. Uvážíme-li, že taktovací kmitočet dnešních procesorů je řádově v GHz, můžeme předpokládat, že se provede 10 milionů základních operací algoritmu za vteřinu (zopakujme, že zde mluvíme o operacích algoritmu, ne o operacích procesoru). Tento odhad je dostatečně konzervativní (opatrný) na to, aby reálně odpovídal většině algoritmů, které budou v tomto studijním materiálu probrány. Následující tabulka ukazuje potřebný čas pro různé časové složitosti a různé počty údajů, které algoritmem zpracováváme. Časy jsou pro přehlednost zaokrouhleny.

	Počty údajů					
Složitost	10	100	1 000	10 000	100 000	1 000 000
$log_2(n)$	0.3 μs	0.7 μs	1 μs	1.3 μs	1.6 μs	2 μs
n	1 μs	10 μs	100 μs	1 ms	10 ms	100 ms
$nlog_2(n)$	3 μs	67 μs	1 ms	13 ms	166 ms	2 s

n^2	10 μs	1 ms	100 ms	10 s	17 min.	28 hod.
2^n	102 μs	4.10^{15} roků				

Algoritmy, jímž odpovídají první čtyři složitosti v tabulce, patří mezi algoritmy s polynomickou časovou složitostí. Třída těchto algoritmů zahrnuje všechny algoritmy, jejich časová složitost je přímo vyjádřena polynomem anebo pro jejich funkci časové složitosti existuje polynom, který ji shora ohraničuje. Například pro funkci $log_2(n)$ je tímto polynomem lineární polynom neboť platí

$$\log_2(n) \leq n \quad \text{pro n = 1,2,…}$$

Za prakticky použitelné obecně považujeme algoritmy s polynomickou časovou složitostí.

Úlohy této třídy považujeme z časového hlediska za řešitelné. Jinak řečeno čas pro provedení algoritmů považujeme obecně za přijatelný, ačkoliv už u kvadratického polynomu je pro větší počty údajů čas potřebný pro výpočet dosti citelný. Všechny základní algoritmy obsažené v tomto studijním textu patří do této třídy.

Poslední složitost v tabulce je vyjádřena exponenciální funkcí. Je zřejmé, že hodnota této funkce roste tak drasticky, že i pro poměrně malé počty údajů je potřebný čas tak velký, že je nemožné takovými algoritmy provést výpočet. Tyto algoritmy patří do třídy algoritmů s nepolynomiální časovou složitostí. Mají tu vlastnost, že funkci jejich časové složitosti nelze shora ohraničit žádným polynomem. Algoritmy této třídy považujeme za neřešitelné v přijatelném čase. Existuje řada praktických úloh, které vedou k algoritmům této třídy. Zejména sem patří úlohy na matematických grafech. Nemožnost tyto praktické úlohy řešit výpočtem algoritmů této třídy se v praxi řeší sestavením algoritmů s přijatelnou (polynomickou) časovou složitostí pro tyto úlohy, které neřeší danou úlohu přesně, ale jen přibližně, čímž typicky dávají o něco horší výsledky, než by dal přesný algoritmus s nepolynomickou časovou složitostí.

Průvodce studiem

Skutečnost, že algoritmy pro řešení některých úloh mají exponenciální časovou složitost, nám komplikuje život. Prakticky se použít nedají, musíme pro ně hledat náhradní, přibližné algoritmy. Obecně je to tedy nepříznivý jev. Má to ale i jednu kladnou stránku – tyto algoritmy lze velmi efektivně využít k ochraně informací pomocí šifrování.

Kontrolní otázky

1. *Jakým způsobem vyjadřujeme časovou složitost algoritmu?*
2. *Vysvětlete, jaký je základní rozdíl z pohledu prakticky využitelnosti mezi algoritmy s polynomickou časovou složitostí a algoritmy s nepolynomickou časovou složitostí.*

Cvičení

1. Pro řešení úlohy máme dva algoritmy, jejichž časové složitosti jsou
   a) $\log_2(n)$
   b) $\sqrt{n}$

   Který z nich je rychlejší, tj. má příznivější časovou složitost?
2. Pro daný algoritmus jsem odvodili, že jeho časová složitost je dána funkcí
   $$n \log_2(n^2) + 1$$
   Když o tomto algoritmu napíšeme, že jeho časová složitost je

1. $\Theta(n\ln(n))$ (*ln* označuje přirozený logaritmus)
2. $\Theta(n\log_{10}(n))$
3. $\Theta(n)$

   které z těchto možností jsou správně?

Úkoly k textu

Představme si, že škrábeme brambory na přípravu oběda pro děti v letním táboře. Základní operací zde bude oškrábání jednoho bramboru. Nechť počet škrábaných brambor pro jeden oběd je 5. Stanovte, jaká je časová složitost algoritmu, je-li počet dětí na táboře *n*.
Jak se změní míra časové složitosti, když
- Brambory nebudu škrábat sám, ale budeme to dělat dva
- Každé dítě si samo oškrábe brambory na svůj oběd

Řešení

1. Který algoritmus je rychlejší, stanovíme podle limity podílu jejich časových složitostí. Vzhledem k tomu, že obě funkce časových složitostí neomezeně rostou, vede to k limitě výrazu typu $\frac{+\infty}{+\infty}$. Pro výpočet použijeme l'Hospitalovo pravidlo

   $$\lim_{n\to+\infty}\frac{f(n)}{g(n)} = \lim_{n\to+\infty}\frac{f'(n)}{g'(n)}$$

   Dále pro logaritmus použijeme pravidlo, jež logaritmus o základu *a* převádí na logaritmus o jiném základu *b*

   $$\log_a(n) = \log_b(n) * \log_b(a)$$

   Nyní již můžeme limitu počítat

   $$\lim_{n\to+\infty}\frac{\log_2(n)}{\sqrt{n}} = \lim_{n\to+\infty}\frac{\ln(n)*\ln(2)}{n^{\frac{1}{2}}} = \lim_{n\to+\infty}\frac{(\ln(n)*\ln(2))'}{(n^{\frac{1}{2}})'} = \lim_{n\to+\infty}\frac{\frac{1}{n}*\ln(2)}{\frac{1}{2}n^{-\frac{1}{2}}} =$$

   $$2\ln(2) * \lim_{n\to+\infty}\frac{\frac{1}{n}}{\frac{1}{\sqrt{n}}} = 2\ln(2) * \lim_{n\to+\infty}\frac{1}{\sqrt{n}} = 0$$

   Tedy první algoritmus má lepší časovou složitost a bude proto při výpočtu obecně rychlejší.

2. Nejprve ověříme možnost a). Vypočítáme limitu podílu výchozí časové složitosti a složitosti uvedené v možnosti a):

   $$\lim_{n\to+\infty}\frac{n\log_2(n^2)+1}{n\ln(n)} = \lim_{n\to+\infty}\frac{2n\log_2(n)+1}{n\ln(n)} = \lim_{n\to+\infty}\frac{2n\ln(n)*\ln(2)+1}{n\ln(n)} =$$

   $$\lim_{n\to+\infty}\left(2\ln(2)+\frac{1}{n\ln(n)}\right) = 2\ln(2) + \lim_{n\to+\infty}\frac{1}{n\ln(n)} = 2\ln(2)$$

Limita je konečná a nenulová, možnost a) je správně. Obdobný způsobem bychom zjistili, že i možnost b) je správně. To víceméně ukazuje, že hodnota základu logaritmu nemá vliv na míru složitosti.
Zbývá možnost c). Opět vypočítáme limitu podílu:

$$\lim_{n\to+\infty} \frac{n \log_2(n^2)+1}{n} = \lim_{n\to+\infty} \left(\log_2(n^2) + \frac{1}{n} \right) = \lim_{n\to+\infty} \log_2(n^2) = +\infty$$

Z ní plyne, že odpověď c) je chybná. Protože hodnota limity podílu je $+\infty$, je míra složitosti algoritmu větší než je míra složitosti lineární funkce *n* uvedené v možnosti c).

2 Datové struktury

Údaje, se kterými algoritmy pracují, mohou být jednak samostatné hodnoty nebo to mohou být strukturované údaje. Samostatné hodnoty jsou tvořeny jednoduchými datovými typy, jako jsou celá čísla, čísla v pohyblivé řádové čárce, řetězce atd. Strukturované údaje mají charakter záznamů. Záznamy většinou popisují nějaké objekty. Je v nich více položek – datových členů, přičemž každý člen popisuje nějaký rys nebo hodnotu daného objektu. Například bude-li záznam popisovat osobu, pak typicky bude obsahovat její rodné číslo, jméno a příjmení, jednotlivé části adresy bydliště (název ulice s popisným číslem, PSČ a název města) atd. Pro popis algoritmů většinou není důležité, pro jaké datové typy budou prakticky použity, zda pro jednoduché nebo pro strukturované datové typy. Budou pracovat s nějakými prvky z nějaké množiny dat. Nicméně aby algoritmy mohly prvky vyhledávat nebo je srovnávat, potřebují, aby u každého prvku byla stanovena hodnota, která ho reprezentuje. U prvků, jež jsou tvořeny jednoduchými datovými typy, je touto hodnotou samotná hodnota prvku. U strukturovaného typu je touto hodnotou většinou nějaký jeho člen (nebo několik jeho členů), a to takový, že jeho hodnota prvek jednoznačně určuje. Hodnotě, která prvek reprezentuje (určuje), budeme říkat klíč prvku.

2.1 Lineární datové struktury

Studijní cíle: Popsat, jaké máme k dispozici lineární datové struktury, a vysvětlit, kdy a kterou z nich použijeme pro uložení hodnot.

Klíčová slova: Pole, seznam, ukazatel, zásobník, fronta.

Potřebný čas: 1 hodina

Lineární datové struktury se vyznačují tím, že jednotlivé prvky množiny dat jsou v nich uloženy postupně za sebou. Tj. každý prvek, který není prvním prvkem, má právě jednoho bezprostředního předchůdce a naopak každý prvek vyjma posledního má právě jednoho bezprostředního následníka. Nejjednodušší způsob lineárního uložení je pole.

Průvodce studiem

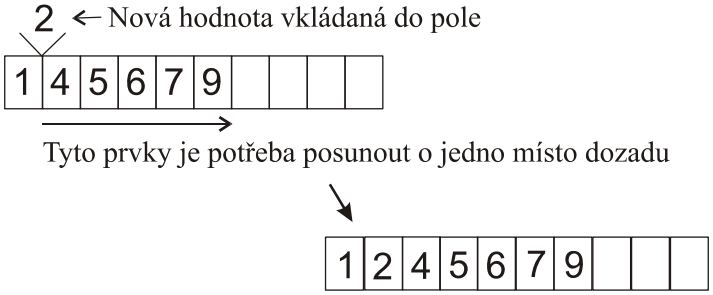
Paměť počítače má lineární uspořádání. Proto pole je nejpřirozenější a nejsnadněji realizovatelná datová struktura pro uložení údajů v paměti.

2.1.1.1 Pole

Pole je velmi často používaná lineární datová struktura.

Pole mají tu velmi příznivou vlastnost, že jsou dostupné ve všech běžných programovacích jazycích. Pole jsou charakterizována svým rozsahem (počtem prvků, které lze do nich uložit), počtem rozměrů (jednorozměrné pole, dvourozměrné pole atd.) a dále způsobem indexování prvků (zda jsou indexovány od 0, od 1 nebo lze počáteční hodnotu indexu zvolit). Pro lineární uložení potřebujeme ten nejjednodušší případ pole, kterým je jednorozměrné pole. Velkou výhodou pole je, že ke každému prvku pole máme přímý přístup pomocí indexu. Použití pole je velmi efektivní, pokud jsou splněny určité předpoklady. Jednak potřebujeme vědět, kolik hodnot do pole budeme ukládat. Dále pokud potřebujeme hodnoty do pole ukládat postupně, budeme to dělat tak, že je budeme ukládat na konec již uložených hodnot. Stejně tak, pokud potřebujeme odstranit hodnoty z pole, pak jen ty, které jsou na konci pole. Pokud nevíme, kolik

hodnot budeme do pole ukládat, může se nám stát, že dojde k vyčerpání pole. To lze řešit použitím tzv. dynamicky alokovaných polí, u nichž lze provést zvětšení jejich rozsahu. Nicméně tato operace je typicky spojena s realokací paměti pro pole (umístění pole v jiné části paměti, kde je volný prostor požadované velikosti) , což s sebou nese přesun již uložených hodnot v poli na nové místo v paměti. To je časově poměrně náročná operace a proto se snažíme, aby ke zvětšování pole docházelo co nejméně. Další nepříjemný problém nastává v případě, kdy hodnoty máme v poli nějakým způsobem uspořádané a potřebujeme vložit další hodnotu tak, aby uspořádání bylo zachováno. Tedy nikoliv na konec, ale někde mezi již uložené hodnoty. V tom případě musíme všechny hodnoty od místa vložení posunout o jedno místo dozadu, aby se uvolnilo místo pro nově vkládanou hodnotu, což je opět často časově náročné.

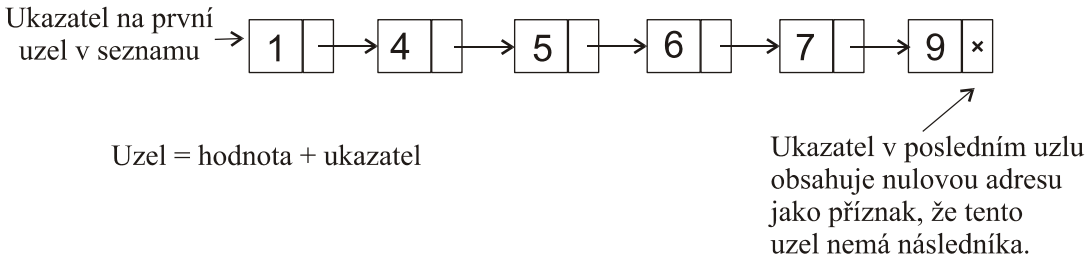


Obdobný problém nastane, pokud z pole odstraňujeme hodnotu, jež v něm není poslední. Pak naopak hodnoty za ní musíme v poli posunout o jedno místo dopředu.

Pokud příslušný algoritmus vyžaduje častější provádění operací, které jsou v poli neefektivní (zvětšení rozsahu, vládání doprostřed uložených hodnot, odstraňování z míst, jež jsou uprostřed uložených hodnot), pak je účelnější pro uložení hodnot místo pole použít lineární dynamickou datovou strukturu – seznam.

2.1.1.2 Seznam

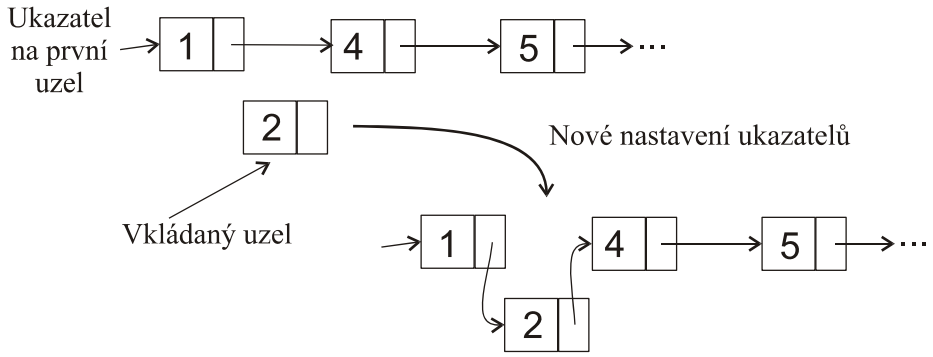
Seznam je lineární dynamická datová struktura. Přívlastek dynamická zde vyjadřuje skutečnost, že je budován postupně, paměť pro něj není na rozdíl od pole přidělena najednou, ale je alokována samostatně pro každý ukládaný datový prvek. Tím na rozdíl od pole není na začátku zapotřebí stanovit počet prvků, které budou do seznamu ukládány. A protože je paměť pro každý ukládaný prvek přidělována samostatně, nejsou na rozdíl od pole prvky v paměti uloženy za sebou, ale víceméně nahodile na různých místech. Tím ovšem ztrácíme základní výhodu pole plynoucí z uložení prvků v paměti za sebou a tou je možnost přejít k dalšímu prvku v poli prostým zvýšením hodnoty indexu. Abychom i v seznamu mohli přejít k následujícímu prvku, ukládáme spolu s každým prvkem ještě ukazatel na místo v paměti, kde je uložená následující prvek. Této dvojici prvek + ukazatel v seznamu říkáme uzel. Připomeňme, že pojem prvek zde používáme v širším významu, tedy může to být jedna hodnota nebo i strukturovaný typ (více hodnot). Abychom mohli se seznamem vůbec pracovat, musíme si navíc někde uschovat ukazatel na první uzel v seznamu.



Termín *ukazatel* je používán v programovacích jazycích pro označení proměnné, do které ukládáme adresu nějakého místa v paměti. U seznamu máme ukazatel na první uzel v seznamu, což je adresa prvního uzlu seznamu v paměti, a každý uzel obsahuje ukazatel na následující uzel, tedy adresu následujícího uzlu v paměti.

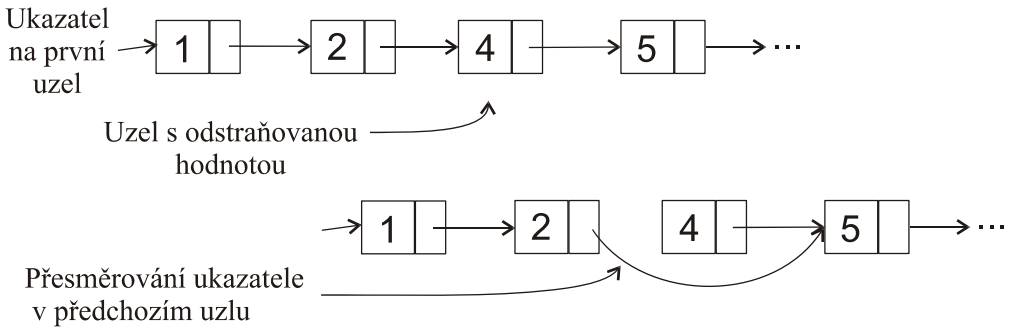
Do seznamu lze snadno kamkoliv vkládat nové prvky.

Vložit nový prvek na libovolné místo v seznamu je na rozdíl od pole poměrně jednoduché. Vytvoříme nový uzel, do něho dáme přidávaný prvek, následně najdeme v seznamu místo vložení, v předchozím uzlu ukazatel přesměrujeme na vkládaný nový uzel a ukazatel v novém uzlu nastavíme na následující uzel v seznamu.

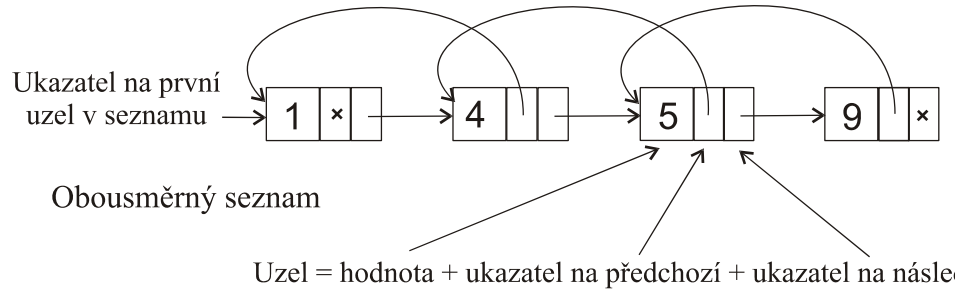


V seznamu lze snadno odstranit kterýkoliv prvek.

Odstranit prvek ze seznamu je ještě jednodušší. Ukazatel v předchozím uzlu přesměrujeme na uzel, jež v seznamu následuje za uzlem s rušeným prvkem.



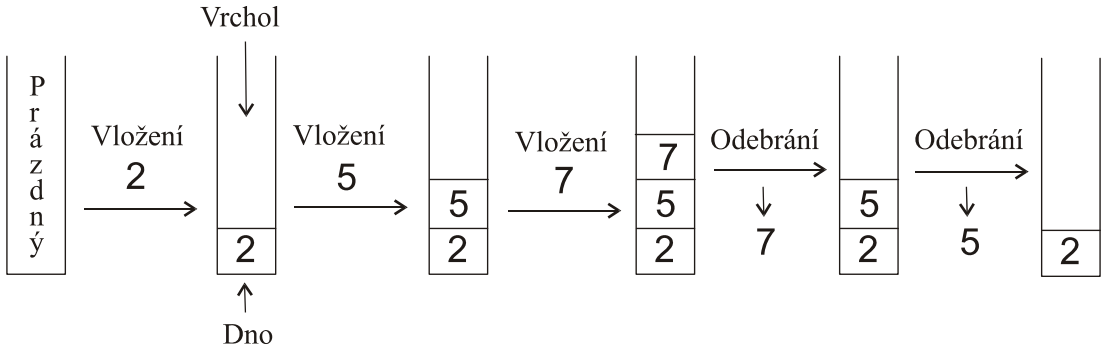
Zatím jsme spíše mluvili o výhodách seznamu. Nyní je čas podívat se na jeho nevýhody. Na rozdíl od pole zde není přímý přístup k libovolnému uloženému prvku. U pole pokud víme, kde je příslušný datový prvek uložen, poměrně snadno vypočítáme index a pomocí něho se okamžitě dostaneme k danému prvku. V seznamu i když víme, v kterém jeho uzlu je daný datový prvek uložen, nám to není nic platné. Nemáme k dispozici ukazatel na tento uzel. Máme jen ukazatel na první uzel v seznamu. Nezbývá než projít postupně uzly seznamu od prvního uzlu až k uzlu s požadovaným prvkem. Navíc pokud už v nějakém uzlu jsme a chceme třeba přejít k prvku v předcházejícím uzlu, musíme opět začít seznam procházet od prvního uzlu, protože v současném uzlu máme jen ukazatel na následující uzel, nikoliv na předchozí. tj. seznam je běžně jen jednosměrný. Proto se někdy používá obousměrný seznam, který v uzlu má dva ukazatele, jeden ukazuje na předchozí uzel a druhý na následující uzel. To umožňuje pohyb po seznamu oběma směry.



Závěrem můžeme zhodnotit, že práce se seznamem je viditelně komplikovanější než s polem. Musíme vytvářet uzly, udržovat ukazatele atd. Navíc nemáme přímý přístup k jednotlivým datovým prvkům uloženým v seznamu, což je v mnoha algoritmech značná nevýhoda. Proto seznamy používáme mnohem méně než pole a to jen v těch případech, kdy použití pole je problematické.

2.1.1.3 Zásobník

Velmi významnou a hojně používanou dynamickou datovou strukturou je zásobník. Je opět lineární datovou strukturou. Jeden konec této struktury je fixní a označujeme ho jako dno zásobníku. Druhý konec tvoří vrchol zásobníku. Zásobník má definovány dvě základní operace: uložení datového prvku na zásobník a odebrání prvku ze zásobníku. Operace uložení probíhá tak, že datový prvek se uloží (přidá) na vrchol zásobníku. Operace odebrání naopak odebere prvek z vrcholu zásobníku. Tudíž operace na zásobníku pracují výlučně s jeho vrcholem. Ostatní datové prvky uložené na zásobníku kromě prvku na vrcholu zásobníku nejsou přímo dostupné. Na začátku je zásobník prázdný, tj. jeho vrchol je totožný s jeho dnem. Postupným přidáváním prvků na zásobník se jeho vrchol posouvá směrem ode dna, naopak odebíráním prvků se vrchol posouvá směrem ke dnu. Je zřejmé, že prvky jsou ze zásobníku odebírány v opačném pořadí, než v jakém byly do něho ukládány. Jde o datovou strukturu typu LIFO (Last In First Out = Poslední dovnitř - první ven).



Zásobník lze vytvořit pomocí pole.

Zásobník patří mezi abstraktní datové struktury. U abstraktních datových struktur definujeme jen jejich vlastnosti a operace, nezabýváme se přitom jejich implementací. Pokud bychom zásobník v praxi potřebovali implementovat, nejsnadnější je to pomocí pole. Začátek pole bude tvořit dno zásobníku, poslední zaplněný prvek pole bude vrcholem zásobníku. Jediným problematickým rysem zde může být stanovení velikosti pole tak, aby nedošlo k přeplnění zásobníku.

Jméno pole, kterým budeme implementovat zásobník, zvolme *z* a předpokládejme, že je indexováno od 0 (hodnota indexu prvního prvku v poli je 0). Dále nechť *n* označuje velikost pole (počet prvků v poli) a proměnná *i* reprezentuje index prvku, který tvoří vrchol zásobníku. Operace budou:

Počáteční nastavení prázdného zásobníku:

```
i = -1;
```

Operaci přidání datového prvku na zásobník ukazují následující příkazy. Nejprve ověříme, zda zásobník není již zaplněn.

```
if (i==n-1) { /* zásobník je plný, nutno něco s tím udělat */ }
else
{ i = i+1;
  z[i] = datový_prvek; }
```

Další příkazy ukazují operaci odebrání datového prvku ze zásobníku a jeho uložení do nějaké proměnné. Nejprve ale ověříme, zda vůbec nějaký prvek je na zásobníku uložen.

```
if (i==-1) { /* zásobník je prázdný, nelze z něho odebrat */ }
else
{ proměnná = z[i];
  i=i-1; }
```

Test, zda zásobník je prázdný, je zde uveden jakou součást operace odebrání prvku ze zásobníku. Někdy v popisech zásobníku bývá tento test uváděn jako další, samostatná operace nad zásobníkem.

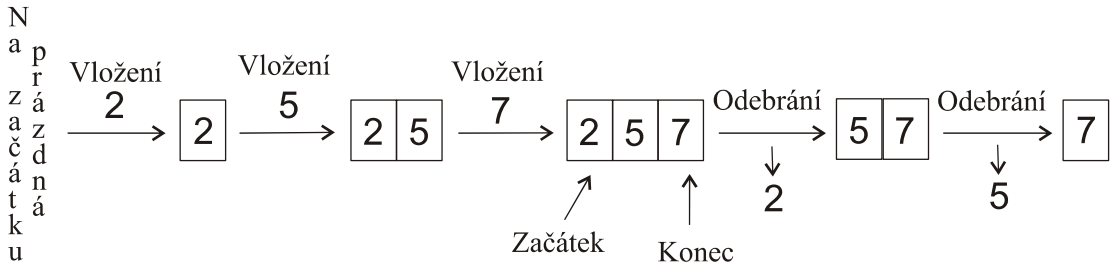
Zásobník lze rovněž implementovat pomocí seznamu. Jako dno je účelné zvolit konec seznamu a tudíž vrcholem bude uzel na začátku seznamu, čímž tento ukazatel je zároveň ukazatelem na vrchol zásobníku. Implementace pomocí seznamu je méně výhodná a efektivní, nicméně zde nehrozí situace, že by došlo k zaplnění zásobníku, jako je to v případě pole.

Průvodce studiem

Pod pojmem zásobník si často představíme část zbraní, ve které jsou uloženy náboje. To, že odebírání nábojů probíhá v opačném pořadí, než byly do něho vkládány, v tomto případě ale není nijak významné.

2.1.1.4 Fronta

Velmi významnou dynamickou datovou strukturou, i když ne tak častou používanou jako zásobník, je fronta. Je také lineární datovou strukturou a má opět definovány dvě základní operace: vložení prvku do fronty a odebrání prvku z fronty. Operace vložení probíhá tak, že datový prvek se uloží na konec fronty. Operace odebrání naopak odebere prvek ze začátku fronty. Na začátku je fronta prázdná. Postupným přidáváním prvků do fronty se její konec vzdaluje od začátku, naopak odebíráním prvků se začátek přibližuje ke konci. Je zřejmé, že prvky jsou z fronty odebírány ve stejném pořadí, v jakém byly do fronty vkládány. Jde o datovou strukturu typu FIFO (First In First Out = První dovnitř - první ven).



Fronta je rovněž abstraktní datová struktura. V jejím popisu opět není uvedeno, jak ji implementovat. V praxi, pokud budeme potřebovat použít frontu, lze ji také vytvořit pomocí

Frontu lze také uložit do pole.

pole, i když ne tak snadno jako zásobník. Problém je v tom, že při odebírání se začátek fronty v poli postupně posunuje dozadu. To vyřešíme cyklickým přechodem z posledního prvku pole na jeho první prvek. Dalším problematickým rysem je zde stejně jako u zásobníku stanovení velikosti pole tak, aby nedošlo k přeplnění fronty.

Označme pole použité pro implementaci fronty opět jménem *z*, počet prvků v něm nechť je *n* a indexování prvků pole nechť je opět od 0. Dále nechť proměnná *i* reprezentuje index prvku pole, který tvoří začátek fronty, a proměnná *j* je indexem prvku pole, jež je koncem fronty, a proměnná *m* obsahuje počet prvků uložených ve frontě. Operace budou:

Počáteční nastavení prázdné fronty:

```
i = 0;
j = 0;
m = 0;
```

Operace přidání nového datového prvku do fronty :

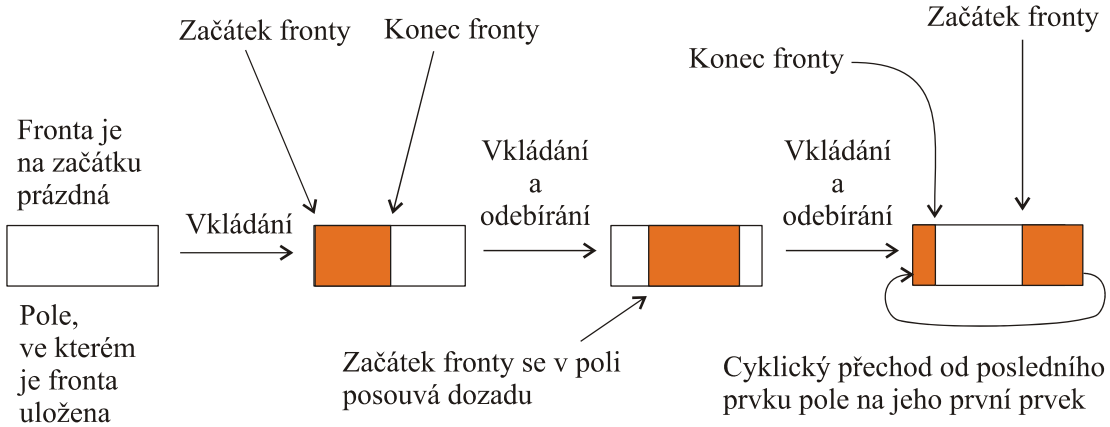
```
if (m==n) { /* fronta je plná */ }
else
{ z[j] = datový_prvek;
  j = (j+1) mod n;
  m = m+1; }
```

Operace odebrání datového prvku z fronty a jeho přesun do nějaké proměnné:

```
if (m==0) { /* fronta je prázdná, nelze z ní odebrat */ }
else
{ proměnná = z[i];
  i = (i+1) mod n;
  m = m-1; }
```

Kde operace *mod* označuje zbytek po dělení.

Posunutí začátku a konce fronty v poli, ve kterém je fronta uložena, při ukládání a odebírání prvků ukazuje následující obrázek.



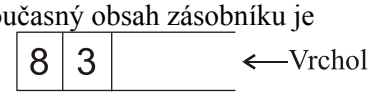
I frontu lze implementovat pomocí seznamu. V tomto případě si kromě ukazatele na první uzel v seznamu budeme rovněž uchovávat ukazatel na poslední uzel v seznamu, abychom při každém přidání do fronty nemuseli seznam procházet od začátku až po jeho konec.

Kontrolní otázky

1. *Kdy použijeme pole a kdy seznam?*
2. *Jaký je hlavní rozdíl mezi zásobníkem a frontou?*

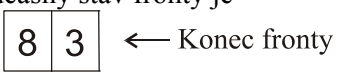
Cvičení

1. Současný obsah zásobníku je

   Nyní provedeme operace
   Odebrání
   Vložení 5
   Vložení 2
   Odebrání
   Vložení 5
   Jaký bude výsledný obsah zásobníku?

2. Současný stav fronty je

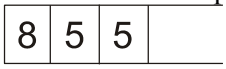
   Nyní s ní provedeme stejné operace jako v předchozím cvičení se zásobníkem. Jaký bude výsledný obsah fronty?

Úkoly k textu

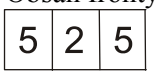
Rozšiřte uvedené implementace zásobníku a fronty pomocí pole o kontrolu přeplnění zásobníku nebo fronty při přidávání prvku a kontrolu prázdného zásobníku nebo fronty při odebírání prvku.

Řešení

1. Obsah zásobníku po uvedených operacích je:

2. Obsah fronty po uvedených operacích je:

2.2 Stromy

Studijní cíle: Zavést základní pojmy, jimiž se v teorii grafů popisují stromy, a dále vysvětlit, jak je lze využít pro uložení datových prvků v algoritmech a jaké jsou jejich výhody ve srovnání s lineárními datovými strukturami.

Klíčová slova: Strom, uzel, list, následník, předchůdce.

Potřebný čas: 1 hodina.

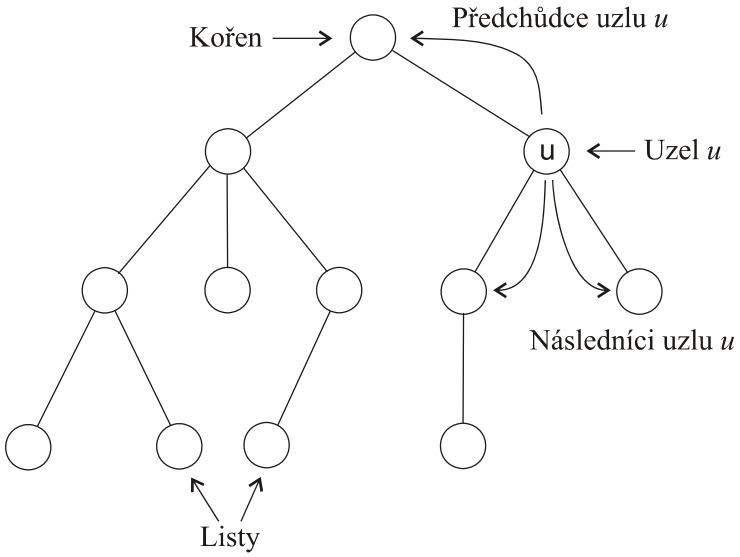
Strom je acyklický graf.

Z nelineárních datových struktur jsou v algoritmech nejpoužívanější stromy. Stromy jsou specifickým případem matematických grafů. Terminologií grafů bychom je popsali jako obyčejné acyklické grafy. Skládají se z uzlů a hran. V uzlech jsou při použití stromů v algoritmech uloženy datové prvky, nad kterými algoritmus probíhá. Hrany reprezentují vztahy mezi uzly (prvky), které jsou základem daného algoritmu.

Strom se kreslí směrem shora-dolů (někdy také zleva-doprava, je-li příliš široký). Zcela nahoře je první uzel stromu, který se nazývá kořen. Pod ním jsou uzly, které jsou jeho následníci. Jsou

s ním spojeny hranami. Tyto uzly mohou mít rovněž následníky, které jsou opět pod nimi a jsou s nimi opět spojeny hranami. Uzly, které nemají žádné následníky, nazýváme listy. Každý uzel vyjma kořene je hranou spojen s právě jedním uzlem, který je nad ním. Tento uzel je jeho předchůdce.

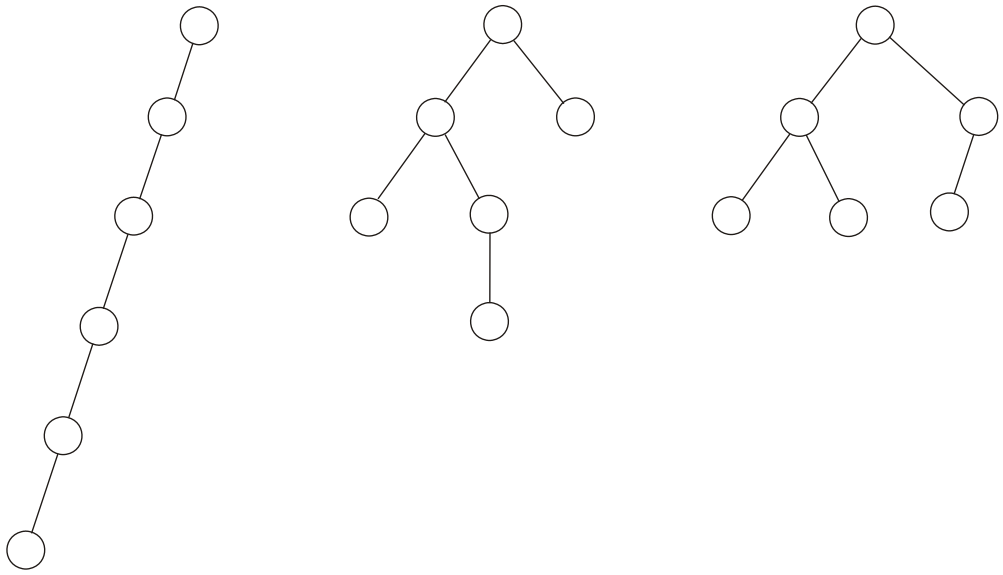
Na kreslení uzlů a hran nejsou žádná zvláštní omezení. Uzly většinou kreslíme jako kružnice, hrany jako rovné čáry.



Nejběžnější jsou binární stromy.

Běžně používané stromy mají zpravidla omezeno, kolik může mít uzel následníků. V tomto ohledu nejjednodušší stromy jsou stromy, v nichž každý uzel může mít nejvýše dva následníky. Používají se poměrně často a říkáme jim binární stromy.

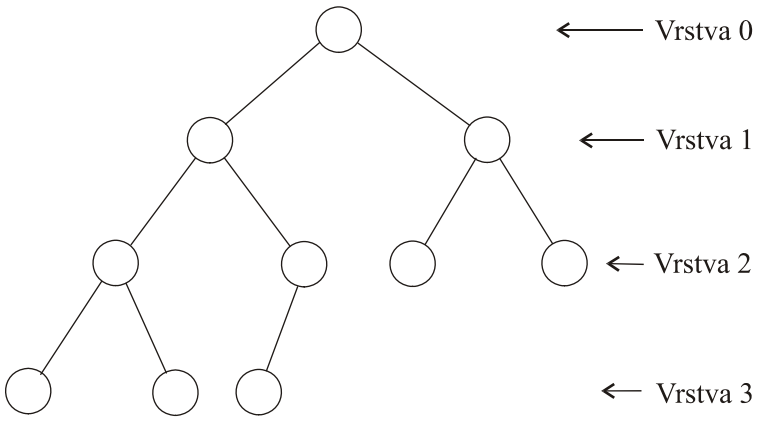
Při použití stromů v algoritmech si uchováváme ukazatel na kořenový uzel. K libovolnému uzlu se pak dostaneme tak, že začneme od kořene a postupně po hranách přecházíme k nižším uzlů, až dojdeme k žádanému uzlu. Přechod od nějakého uzlu po hraně k jeho následníkovi považujeme za jednu základní operaci. Počet operací potřebný k tomu, abychom se od kořene dostali k danému uzlu, je roven počtu hran, které jsou na cestě od kořene k tomuto uzlu. Tento počet nazýváme vzdáleností uzlu od kořene. Maximum ze vzdáleností uzlů od kořene stromu nazveme výškou stromu. Výška stromu je tedy vzdálenost kořene od listů, které jsou ve stromu nejníže. Zřejmě čím má strom při daném počtu uzlů menší výšku, tím je to výhodnější, neboť tím nižší je maximální délka cesty od kořene k uzlům stromu. Na následujícím obrázku jsou tři binární stromy. Všechny mají stejný počet uzlů – šest.



Levý strom je nejméně výhodný. V podstatě je to seznam a o seznamu víme, že časová složitost přístupu k jeho uzlům je lineární. Naproti tomu strom zcela vpravo má tu vlastnost, že má při daném počtu uzlů nejmenší možnou výšku. Je to vyvážený binární strom.

Výška stromu je důležitá pro efektivnost algoritmů.

Vyvážený binární strom je takový binární strom, který má ve všech vrstvách maximální možný počet uzlů vyjma poslední vrstvy, která může být zaplněna jen zčásti. Vrstvou přitom tady rozumíme všechny uzly, které mají stejnou vzdálenost od kořene, tedy mají stejnou vodorovnou úroveň při běžném nakreslení stromu shora-dolů. Tuto vzdálenost nazveme číslem vrstvy. Podíváme-li se na následující příklad vyváženého binárního stromu



můžeme z něho odvodit, jaké počty uzlů budou obecně v jednotlivých vrstvách vyváženého binárního stromu výšky *h*.

Číslo vrstvy	Počet uzlů v ní
0	1
1	2
2	2^2
…	…
h-1	2^{h-1}
h	1 .. 2^h

Odtud pro počet uzlů *n* v celém stromu dostáváme vztah

$$1+2+2^2+\ldots+2^{h-1}+1 \le n \le 1+2+2^2+\ldots+2^{h-1}+2^h$$

Jeho postupnými úpravami (použitím součtu geometrické řady)

$$2^h \le n \le 2^{h+1}-1$$

$$\frac{n+1}{2} \le 2^h \le n$$

$$\log_2\left(\frac{n+1}{2}\right) \le h \le \log_2(n)$$

$$\log_2(n+1)-1 \le h \le \log_2(n)$$

Z tohoto plyne důležitý závěr, že ve vyváženém binárním stromu výška stromu logaritmicky závisí na počtu uzlů v něm, což můžeme vyjádřit jako složitost *Θ(log(n))*.

Průvodce studiem

Jak uvidíme dále, stromy jsou základem velmi efektivních algoritmů zejména pro vyhledávání.

1. *Který z uzlů stromu je kořen a který list?*
2. *Co je vyvážený binární strom?*

Cvičení

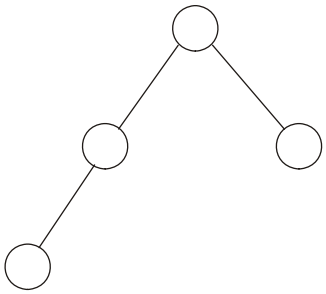
1. Jakou výšku má vyvážený binární strom se 100 uzly?
2. Co kdybychom oslabili požadavek na vyvážený strom tak, že bychom požadovali jen minimální výšku. Mohlo by to mít vliv na průměrnou vzdálenost uzlů stromu od kořene? Tj. mohl by nastat případ, že bychom mohli mít strom se stejnou minimální výškou a přitom s jiným průměrem vzdáleností od kořene, než má vyvážený strom dle definice uvedené v této kapitole. (Průměrná vzdálenost od kořene je součet vzdáleností všech uzlů od kořene dělený celkovým počtem uzlů ve stromu. Do celkového počtu uzlů je zahrnut i kořen. Jeho vzdálenost od kořene je přirozeně nula.)

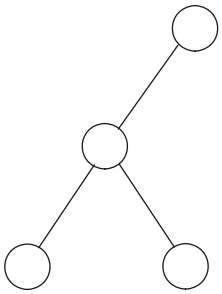
Úkoly k textu

Odvoďte závislost výšky stromu na počtu uzlů pro vyvážený strom, který je *r*-ární. Kde *r* označuje, kolik následníků může mít každý uzel stromu ($r \ge 2$).

Řešení

1. Výška stromu je 6.
2. Následující obrázek ukazuje vlevo vyvážený binární strom se 4 uzly s průměrnou vzdáleností uzlů od kořene 1. Vpravo je binární strom se stejným počtem uzlů a se stejnou výškou, ale s průměrnou vzdáleností uzlů od kořene 1.25. Z toho lze usoudit, že pokud bychom nepožadovali, aby vyvážený strom měl neúplnou jen poslední vrstvu, mělo by to nepříznivý vliv na průměrnou vzdálenost uzlů od kořene.





Průvodce studiem

Popis algoritmů začneme tříděním.Pak budou následovat algoritmy vyhledávání. I když v běžném životě častěji něco hledáme než třídíme…

3 Třídění

Studijní cíle: Definovat úlohu třídění a dále zavést pojmy jednak pro popis vlastního třídícího problému a dále pro klasifikaci a popisy algoritmů třídících metod.

Klíčová slova: Uspořádání, klíč, vnitřní třídění, vnější třídění.

Potřebný čas: 20 minut.

Třídění je seřazení dle velikosti.

Potřeba setřídit údaje se v praxi objevuje velmi často. Proto algoritmy pro třídění patří do skupiny základních, velmi používaných algoritmů. Úloha třídění spočívá v seřazení prvků datové množiny vzestupně podle jejich velikosti. Máme tedy *n* prvků dat

$$a_1, a_2, \ldots a_n$$

a předpokládáme, že jsou takového datového typu, u kterého je definováno srovnání dle velikosti. Úkolem je prvky seřadit do posloupnosti

$$a_{i_1}, a_{i_2}, \ldots a_{i_n}$$

tak, že platí

$$a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$$

Můžeme mít také opačné setřídění – od největší hodnoty po nejmenší, kdy platí

$$a_{i_1} \geq a_{i_2} \geq \ldots \geq a_{i_n}$$

Například setříděním následujících přirozených čísel

7 2 3 1 6 7 3 4 5

dostaneme posloupnost

1 2 3 3 4 5 6 7 7

Pro popis algoritmů není podstatné, zda třídění je vzestupné nebo sestupné. Ve výkladu všech algoritmů v tomto materiálu předpokládáme, že třídíme vzestupně.

Algoritmy třídění lze rozdělit do dvou základních skupin

- Algoritmy vnitřního třídění
- Algoritmy vnějšího třídění

Vnitřní třídění probíhá výlučně ve vnitřní paměti počítače.

Algoritmy vnitřního třídění mají tu základní vlastnost, že během třídění jsou všechny tříděné prvky uloženy ve vnitřní paměti počítače. To je výhodné, protože veškeré operace třídění (srovnání, přesuny) probíhají výlučně operacemi nad hodnotami ve vnitřní paměti.

V algoritmech vnějšího třídění jsou tříděné prvky uloženy v souborech na vnější paměti (pevném disku). Tyto jsou průběžně v malých počtech přesouvány do vnitřní paměti, zde jsou nad nimi provedeny příslušné operace (srovnání, přesuny) a následně jsou opět ukládány do souborů na vnější paměť. Vnější třídění je obecně pomalejší než vnitřní třídění, protože operace čtení a zápisu na vnější paměť jsou výrazně pomalejší než operace prováděné jen ve vnitřní paměti. Vnější třídění proto používáme výlučně v těch případech, kdy tříděných údajů je tolik, že se najednou nevejdou do paměti a nelze pro ně použít vnitřní třídění.

3.1 Vnitřní třídění

Prvky při třídění vkládáme nebo vyměňujeme nebo vybíráme.

Algoritmy třídění jsou založeny na dvou základních operacích – srovnání a přesunech. Podle toho, jak tyto přesuny probíhají, dělíme algoritmy třídění do tří základních skupin:

- Třídění vkládáním
- Třídění výměnou
- Třídění výběrem

U třídění vkládáním vytváříme setříděnou posloupnost tak, že ji postupně zvětšujeme vkládáním dalších prvků na příslušná místa.

U třídění výměnou vytváříme setříděnou posloupnost tak, že postupně vzájemně zaměňujeme dva vybrané prvky, až tím nakonec dosáhneme úplného setřídění.

U třídění výběrem postupně vybíráme prvky ze vstupní posloupnosti a přidáváme je k vytvářené setříděné posloupnosti.

Pro tyto postupy existují intuitivně jednoduché postupy, které vzhledem ke své jednoduchosti mají větší časovou složitost, a sofistikovanější postupy, které vedou ke komplikovanějším algoritmům, nicméně jejich časová složitost je výrazně lepší.

Kontrolní otázky

1. *Kdy použijeme vnitřní třídění a kdy vnější třídění?*
2. *Vysvětlete, kdy třídící metodu nazýváme tříděním vkládáním, kdy tříděním výměnou a kdy tříděním výběrem.*

3.2 Přímé metody třídění

Nejprve probereme jednoduché metody třídění. Ty jsou označovány jako přímé metody třídění.

Studijní cíle: Popsat algoritmy jednotlivých přímých metod vnitřního třídění. Demonstrovat, jak vlastní třídící proces probíhá a dále popsat, jak tyto algoritmy prakticky implementovat.

Klíčová slova: Srovnání, vložení, výběr, výměna, bublinkové třídění.

Potřebný čas: 3 hodiny

3.2.1 Třídění přímým vkládáním

Předpokládáme, že tříděné prvky máme na začátku uspořádány v nějakou posloupnost. V praxi při implementaci metod třídění používáme pro uložení prvků pole. Výchozí posloupnost nám tedy tvoří počáteční uložení prvků v poli.

Počet tříděných prvků nechť je n. Prvky si označme

$$a_1, a_2, \ldots a_n$$

Třídění probíhá vkládáním prvků na jejich cílové místo.

Pole, ve kterém jsou uloženy prvky, je v průběhu třídění rozděleno na dvě části - setříděnou část (ta je první) a nesetříděnou část. V každém kroku vezmeme první prvek v nesetříděné části, projdeme setříděnou část, najdeme v ní místo vložení a na toto místo prvek vložíme. Vložení se provede tak, že posuneme všechny prvky počínaje místem vložení až po konec setříděné části o jednu pozici dozadu, aby se uvolnilo místo pro vkládaný prvek, a na uvolněnou pozici nový prvek vložíme. Hledání pozice vložení lze udělat postupným procházením od začátku setříděné

části a srovnáváním vkládaného prvku s prvky setříděné části. Výhodnější ale v tomto případě je zvolit opačný směr, začít procházení od konce setříděné části, neboť můžeme přitom souběžně dělat i posouvání prvků setříděné části o jednu pozici dozadu. Z této možnosti vychází následující algoritmus.

Popis algoritmu

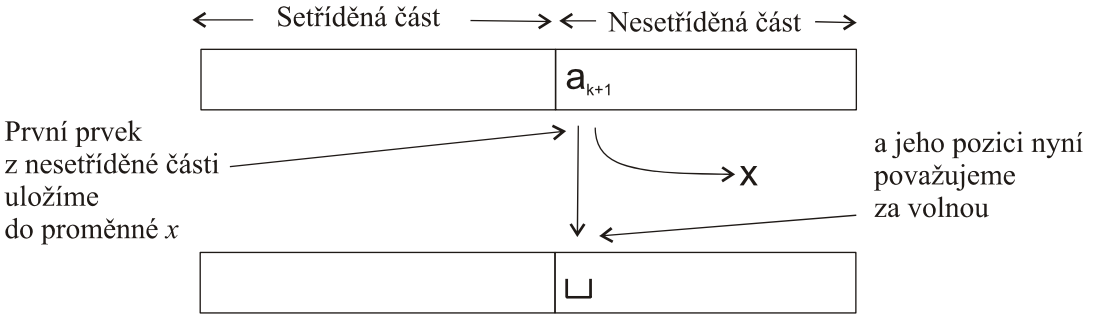
1. Počáteční krok

Na začátku vytvoříme počáteční rozdělení pole na setříděnou a nesetříděnou část. Setříděnou částí bude první prvek v poli, nesetříděnou částí bude zbývajících *n*-1 prvků.

2. Průběžný krok

Procházíme setříděnou část od konce, až najdeme místo vložení.

Nechť setříděná část je nyní tvořena *k* prvky a za ní je zbývajících *n-k* prvků, které tvoří nesetříděnou část. Vezmeme první prvek z nesetříděné části (a_{k+1}), uložíme ho do pomocné proměnné, označme ji *x,* a pozici tohoto prvku považujeme za volnou.



Nyní odzadu procházíme prvky v setříděné části a postupně pro indexy

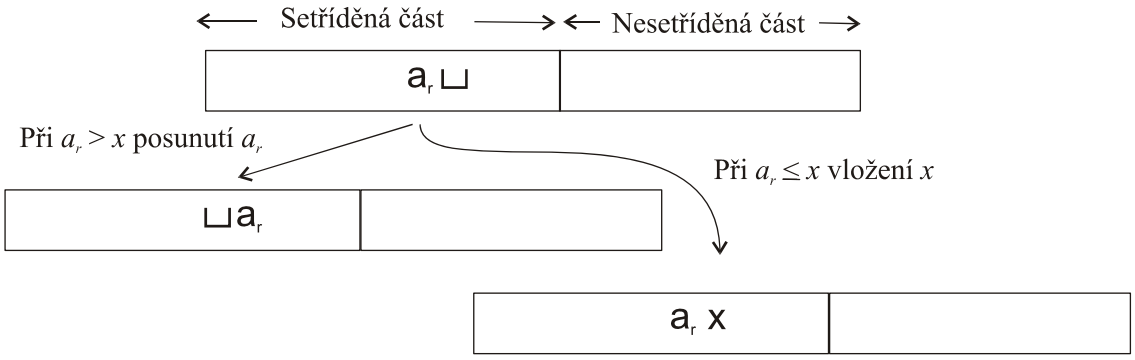
$r = k, k-1, k-2, \ldots$

srovnáváme, zda mezi prvkem a_r a uloženým prvkem *x* platí

$$a_r > x \ .$$

Pokud ano, posuneme prvek a_r o jednu pozici dozadu (tato pozice za prvkem je v tomto okamžiku volná) a jeho původní pozice se nyní stane novou volnou pozicí.

Celý proces srovnání skončí v okamžiku, kdy buďto pro některý prvek platí $a_r \leq x$ anebo celá setříděna část je už porovnána (a rovněž i posunuta dozadu). Pozice vložení ve chvíli, kdy ukončíme srovnávání, odpovídá stávající volné pozici. Na ni uložíme prvek obsažený v proměnné *x*.



Po každém provedení průběžného kroku se zvětší velikost setříděné části o jeden prvek, až nakonec setříděná část bude zahrnovat všechny prvky.

Příklad. Mějme setřídit přirozená čísla

7 1 2 8 4 5 3 9

Po prvním kroku bude pole rozděleno na dvě části:

7 | 1 2 8 4 5 3 9

První prvek v nesetříděné části je číslo 1. Uložíme ho do proměnné *x* a jeho místo uvolníme

7 | • 2 8 4 5 3 9

Srovnáme 7 s *x* → posunutí 7 doprava:

• | 7 2 8 4 5 3 9

Už není co srovnávat, vložíme *x* na volnou pozici a posuneme hranici setříděné části:

1 7 | 2 8 4 5 3 9

Další krok - uložíme první prvek 2 do proměnné *x* a jeho místo uvolníme:

1 7 | • 8 4 5 3 9

Srovnáme 7 s x → posunutí 7 doprava:

1 • | 7 8 4 5 3 9

Srovnáme 1 s *x* → vložení *x* a posunutí hranice setříděné části:

1 2 7 | 8 4 5 3 9

Další krok - uložíme první prvek 8 do proměnné *x* a jeho místo uvolníme:

1 2 7 | • 4 5 3 9

Srovnáme 7 s *x* → vložení *x* a posunutí hranice setříděné části:

1 2 7 8 | 4 5 3 9

Atd.

Složitost metody

Při stanovení složitosti třídění uvažujeme operace srovnání a operace vložení.

Uvedený algoritmus třídění je založen na operacích srovnání a vložení. Vezměme nejprve operaci srovnání. Nechť počet prvků v setříděné části je *k*. Abychom zjistili místo vložení, uděláme 1 až *k* srovnání. Jedno srovnání v případě, kdy se prvek vkládá hned na konec setříděné části, *k* srovnání v případě, kdy prvek patří na první nebo druhou pozici v setříděné části. Odtud pro jeden krok dostáváme

$$\text{Průměrný počet srovnání} = \frac{1+k}{2}$$

Maximální počet srovnání = *k*

Vezmeme-li v úvahu, že délky setříděných částí v jednotlivých krocích jsou $k = 1, 2, \ldots, n\text{-}1$, dostaneme celkové počty srovnání:

$$\text{Průměrný počet srovnání} = \frac{2+3+\ldots+n}{2} = \frac{1+2+3+\ldots+(n-1)}{2} =$$

$$= \frac{\frac{n(n+1)}{2}-1}{2} = \frac{n^2+n-2}{4}$$

$$\text{Maximální počet srovnání} = 1+2+\ldots+(n-1) = \frac{(n-1)\,n}{2} = \frac{n^2-n}{2}$$

Z toho plyne, že operace srovnání má kvadratickou složitost bez ohledu na to, zda uvažujeme průměrný nebo maximální počet srovnání.

Nyní uvažujme operaci přesunu prvku o jednu pozici dozadu. Těch v jednom kroku proběhne 0 až k podle toho, na které místo je prvek vkládán . Odtud pro jeden krok dostáváme

$$\text{Průměrný počet přesunů } = \frac{k+0}{2} = \frac{k}{2}$$

Maximální počet přesunů = k

A pro celé třídění dostaneme

$$\text{Průměrný počet přesunů } = \frac{1+2+...+(n-1)}{2} = \frac{\frac{(n-1)\,n}{2}}{2} = \frac{n^2-n}{4}$$

$$\text{Maximální počet přesunů } = 1+2+...+(n-1) = \frac{(n-1)\,n}{2} = \frac{n^2-n}{2}$$

Z toho plyne, že operace přesunu má kvadratickou složitost bez ohledu na to, zda uvažujeme průměrný nebo maximální počet přesunů.

Časová složitost této metody třídění je kvadratická.

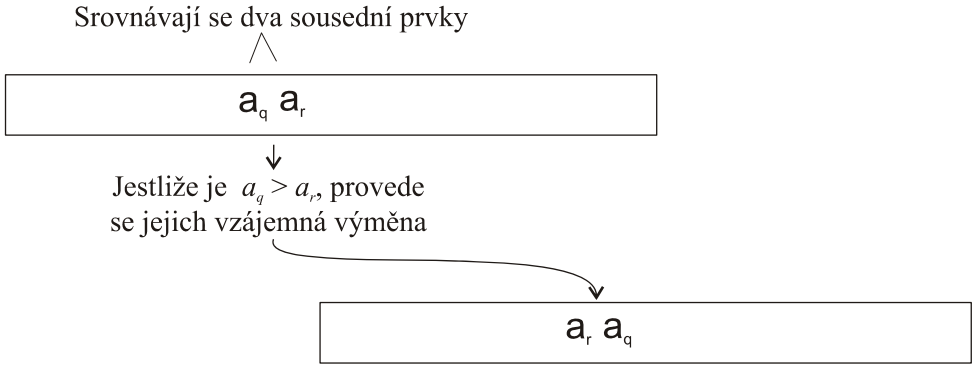
Jestliže algoritmus zahrnuje více operací, výsledná míra složitosti se odvozuje vždy od operace, která má největší složitost. Zde obě operace mají stejnou míru složitosti - kvadratickou.

Závěr: Třídění přímým vkládáním má časovou složitost $\Theta(n^2)$.

3.2.2 Třídění přímou výměnou (bublinkové třídění)

Při třídění přímou výměnou procházíme postupně pole s tříděnými prvky směrem od začátku k jeho konci a srovnáváme sousední dvojice, zda prvky v nich jsou podle velikosti ve správném pořadí. Jestliže ne, tedy větší prvek je před menším, provedeme jejich vzájemnou výměnu.

Třídíme vzájemnou výměnou dvou sousedních prvků.



Snadno se ověří, že největší prvek obsažený v tříděné posloupnosti se těmito výměnami dostane na konec, tedy na své cílové místo, ať byl předtím kdekoliv. V další kroku celý postup zopakujeme, ale už bez posledního prvku, tedy jen s prvními n-1 prvky. Při něm se obdobným způsobem dostane na své cílové místo předposlední prvek setříděné posloupnosti. V následujícím kroku už budeme procházet jen n-2 prvků atd. Je zřejmé, že každým průchodem se délka procházené části sníží o jeden prvek, až nakonec v posledním průchodu bude procházená část mít jen dva prvky a po jejich srovnání a případné výměně je třídění dokončeno.

Příklad. Budeme opět třídit posloupnost

7 1 2 8 4 5 3 9

Postupně srovnáváme sousední prvky

Srovnání 7 s 1 → výměna

1 7 2 8 4 5 3 9

Srovnání 7 s 2 → výměna

1 2 7 8 4 5 3 9

Srovnání 7 s 8 → ve správném pořadí
Srovnání 8 s 4 → výměna

1 2 7 4 8 5 3 9

Srovnání 8 s 5 → výměna

1 2 7 4 5 8 3 9

Srovnání 8 s 3 → výměna

1 2 7 4 5 3 8 9

Srovnání 8 s 9 → ve správném pořadí

V dalším průchodu budeme srovnávat už jen 7 prvků

1 2 7 4 5 3 8 | 9

Srovnání 1 s 2 → ve správném pořadí
Srovnání 2 s 7 → ve správném pořadí
Srovnání 7 s 4 → výměna

1 2 4 7 5 3 8 | 9

Srovnání 7 s 5 → výměna

1 2 4 5 7 3 8 | 9

Srovnání 7 s 3 → výměna

1 2 4 5 3 7 8 | 9

Srovnání 7 s 8 → ve správném pořadí

V dalším průchodu budeme srovnávat už jen 6 prvků

1 2 4 5 3 7 | 8 9

Srovnání 1 s 2 → ve správném pořadí
Atd.

Složitost metody

Složitost metody této metody je kvadratická.

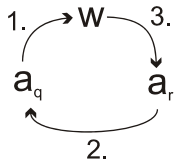
Uvedený algoritmus třídění je založen na operacích srovnání a výměny. Vezměme opět nejprve operaci srovnání. V prvním průchodu se prochází všech n prvků, což reprezentuje srovnání n-1 sousedních dvojic. V druhém průchodu se prochází n-1 prvků, tedy n-2 srovnávaných dvojic. V posledním průchodu má procházená část dva prvky, tedy se provede jen jedno srovnání. Odtud dostáváme

$$\text{Celkový počet srovnání} = (n-1) + (n-2) + \ldots + 1 = \frac{(n-1)\,n}{2} = \frac{n^2 - n}{2}$$

Nyní uvažujme operaci výměn. Výměn se provede maximálně tolik, kolik je srovnání. Minimální možný počet výměn je žádná výměna (prvky jsou na začátku uspořádány tak, že jsou setříděné). Přitom za základní operaci zde nepovažujeme přímo výměnu, ale operace přesunů, pomocí kterých je výměna provedena. Jedna výměna vyžaduje tři přesuny:

1. První prvek přesuneme do pracovní proměnné.

2. Druhý prvek přesuneme na místo prvního prvku.
3. První prvek, který je v současnosti uložený v pracovní proměnné, přesuneme z pracovní proměnné na místo druhého prvku.



Výměna sousedních prvků a_q a a_r s použitím pracovní proměnné w

Odtud dostáváme pro počty přesunů:

$$\text{Průměrný počet přesunů} = \frac{3(n^2 - n)}{4}$$

$$\text{Maximální počet přesunů} = \frac{3(n^2 - n)}{2}$$

Opět obě základní operace (srovnání a přesuny) mají kvadratickou složitost.

Závěr: Třídění přímou výměnou má časovou složitost $\Theta(n^2)$.

Naprogramování je velmi snadné.

Implementace této metody třídění je velmi snadná. Program tvoří dva vnořené cykly. Jejich tělo obsahuje podmíněný příkaz, jež zajišťuje operaci srovnání, a tří příkazy přiřazení, které realizují vzájemnou výměnu sousedních prvků. Pro svoji jednoduchost je třídění přímou výměnou velmi oblíbené. Používá se pro něj název bublinkové třídění. Toto označení je odvozeno od skutečnosti, že prvky postupně jednotlivými výměnami „probublávají“ na svoji cílovou pozici. Byly navrženy i některé modifikace metody, jež zlepšují její efektivitu v určitých situacích. Uveďme dvě nejvýznamnější z nich.

Metoda je známá pod názvem bublinkové třídění.

Vezměme následující posloupnost:

3 1 9 2 8 4 5 7

Stav po prvním průchodu je

1 3 2 8 4 5 7 | 9

Stav po druhém průchodu je

1 2 3 4 5 7 | 8 9

Pokud nedojde k žádné výměně, můžeme třídění ukončit.

Zbývá ještě 5 průchodů s délkami procházených částí 6,5,4,3 a 2. Přitom viditelně všechny jsou zbytečné, neboť posloupnost je v této chvíli už setříděna. Proto algoritmus rozšíříme tak, že v každém průchodu sledujeme, zda v něm došlo vůbec k nějaké výměně. Jestliže ne, třídění se ukončí. V našem příkladu by takto rozšířený algoritmus provedl ještě 3. průchod, ve kterém by se zjistilo, že žádná výměna už během něho neproběhla, a tím by se třídění ukončilo. Tedy zbývající 4 průchody by už vůbec neproběhly.

Vezměme další příklad, z něhož vyplyne, jak ještě dále lze zlepšit vlastnosti metody:

2 3 4 5 7 8 9 1

Stav po prvním průchodu

2 3 4 5 7 8 1 | 9

Stav po druhém průchodu

2 3 4 5 7 1 | 8 9

Někdy je výhodné posloupnost procházet střídavě od začátku a od konce.

I když výchozí posloupnost vypadá pro třídění velmi příznivě, neboť je zapotřebí jen přesunout prvek s hodnotu 1 na začátek, přesto to probíhá velmi pomalu. Je to dáno skutečností, že při procházení směrem od začátku ke konci se prvky s velkými hodnotami rychle dostávají dozadu,

naproti prvky s malými hodnotami postoupí dopředu v každém průchodu jen o jednu pozici. Kdybychom v uvedené posloupnosti použili opačný směr, tedy odzadu směrem dopředu, dostal by se prvek s hodnotou 1 na své místo hned v prvním průchodu. Z toho vychází modifikace metody, ve které se směry procházení v každém průchodu střídají. V prvním průchodu je směr procházení odpředu, čímž se poslední prvek dostane na své místo. V druhém průchodu se prvních *n*-1 prvků prochází směrem odzadu, čímž se první prvek dostane na své místo. Ve třetím průchodu se *n*-2 středních prvků prochází opět směrem odpředu, čímž se předposlední prvek dostane na své místo. Atd.

V našem příkladu bude stav po prvním průchodu

2 3 4 5 7 8 1 | 9

Stav po druhém průchodu

1 | 2 3 4 5 7 8 | 9

Ve třetím průchodu se bude procházet střední posloupnost, přičemž se zjistí, že nedojde k žádné výměně, čímž se po tomto průchodu třídění ukončí.

Poznamenejme na závěr, že ačkoliv tyto modifikace v určitých případech snižují čas třídění, na celkovou časovou složitost metody nemají vliv. Ta zůstává stále kvadratická.

3.2.3 Třídění přímým výběrem

Třídění probíhá vybráním vždy nejmenšího prvku ze zbývajících prvků.

Při třídění přímým výběrem je rovněž v průběhu třídění pole s prvky rozděleno na dvě části. Zde ale část obsahující již setříděné prvky je první a část s doposud nesetříděnými prvky je za ní.

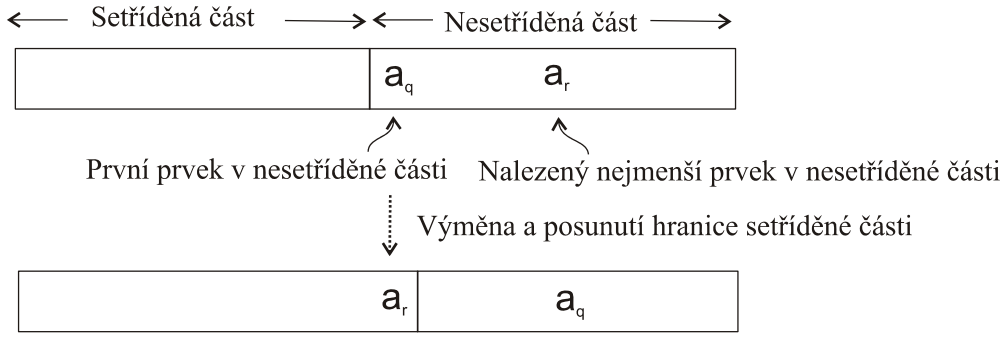
Popis algoritmu

1. Počáteční krok

Na začátku celé pole, tj. všech *n* prvků tvoří nesetříděnou část.

2. Průběžný krok

Nechť setříděná část, která je na začátku, má *n-k* prvků a setříděná část za ní má *k* prvků. V nesetříděné části vybereme nejmenší její prvek. Postupujeme tak, že si na začátku zapamatujeme pozici prvního prvku nesetříděné části. Nyní postupně procházíme prvky nesetříděné části počínaje od jejího druhého prvku a každý z nich srovnáváme s prvkem na zapamatované pozici. Jestliže srovnávaný prvek je menší, jeho pozice se stane novou zapamatovanou pozicí. Je zřejmé, že na konci je na zapamatované pozici nejmenší prvek. Ten musíme dostat na konec setříděné části. To nejjednodušeji uděláme tak, že ho vyměníme s prvním prvkem v nesetříděné části.



Po každém provedení průběžného kroku zvětšíme velikost setříděné části o jeden prvek. Tedy posuneme rozhraní mezi setříděnou a nesetříděnou částí o jednu pozici doprava. V okamžiku,

kdy setříděná část bude obsahovat *n*-1 prvků, je třídění ukončeno, protože v nesetříděné části tímto zůstane už jen největší prvek, který je na konci a tudíž na své cílové pozici.

Příklad. Mějme setřídit přirozená čísla

7 1 2 8 4 5 3 9

Zapamatujeme si pozici prvku 7
Srovnáme s 1 → zapamatujeme si pozici prvku 1
Srovnáme s 2, 8, 4, 5, 3, 9

Zapamatovaná je pozice prvku 1, ten vyměníme s prvním prvkem 7

1 | 7 2 8 4 5 3 9

Další krok – zapamatujeme si pozici prvku 7
Srovnáme s 2 → zapamatujeme si pozici prvku 2
Srovnáme s 8, 4, 5, 3, 9

Zapamatovaná je pozice prvku 2, ten vyměníme s prvním prvkem 7

1 2 | 7 8 4 5 3 9

Atd.

Složitost metody

I tato metoda má kvadratickou časovou složitost.

Uvedený algoritmus třídění je založen na operacích výběru a výměny. Vezměme nejprve operaci výběru. Nechť počet prvků v nesetříděné části je *k*. Abychom našli její nejmenší prvek, potřebujeme k tomu *k*-1 srovnání. Neboť počínaje druhým prvkem v nesetříděné části postupně srovnáváme všechny její prvky až po poslední prvek, vždy s právě zapamatovaným prvkem.

Délky nesetříděných částí jsou v jednotlivých krocích *n*, *n*-1, …,*2*. Odtud dostáváme:

$$\text{Celkový počet srovnání} = (n-1) + (n-2) + ... + 1 = \frac{(n-1)\,n}{2} = \frac{n^2 - n}{2}$$

Nyní uvažujme operaci výměny vybraného, nejmenšího prvku s prvním prvkem nesetříděné části. Ta v každém kroku proběhne jen jednou. Teoreticky vzato když nastane případ, že nejmenší prvek je hned na začátku nesetříděné části, nemusela by tato operace vůbec proběhnout. Nicméně je jednodušší a i efektivnější případně vyměnit prvek sám se sebou, než před každou výměnou ověřovat, zda nenastal případ, že výměna není potřebná. Třídících kroků je *n*-1 a s ohledem na to, že jedna výměna vyžaduje tři přesuny, odtud dostáváme

$$\text{Celkový počet přesunů} = 3\,(n-1)$$

Největší složitost má u této třídící metody operace výběru a ta určuje celkovou složitost metody.

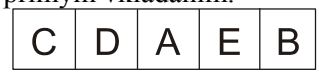
Závěr: Třídění přímým výběrem má časovou složitost $\Theta(n^2)$.

Kontrolní otázky

1. *Jakým způsobem probíhá u metody přímého vkládání nalezení místa, kam má být vložen další nesetříděný prvek?*
2. *Jak by musela výchozí posloupnost vypadat (být uspořádána), aby při bublinkovém třídění po každém srovnání musela být provedena výměna?*
3. *Jakým způsobem je u metody přímého výběru realizováno umístění dalšího vybraného prvku na svoji cílovou (setříděnou) pozici?*
4. *U všech přímých metod je tříděné pole rozdělené na dvě části – již setříděnou a doposud nesetříděnou. Co kdybychom třídili v opačném pořadí (od největšího prvku k nejmenšímu). Mělo by to vliv na vzájemnou pozici setříděné a nesetříděné části?*

Cvičení

1. Po jednotlivých krocích setřiďte podle abecedy následujících pět písmen metodou třídění přímým vkládáním.



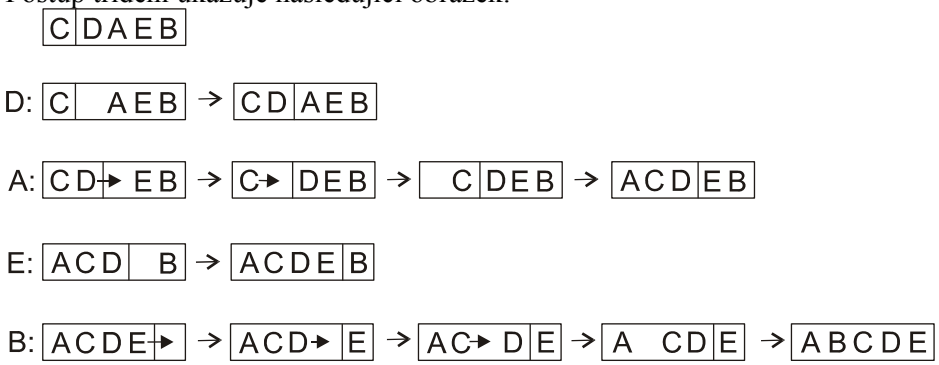
2. Stejnou posloupnost jako v předchozím cvičení setřiďte bublinkovým třídění.
3. Stejnou posloupnost jako v první cvičení setřiďte metodou třídění přímým výběrem.

Úkoly k textu

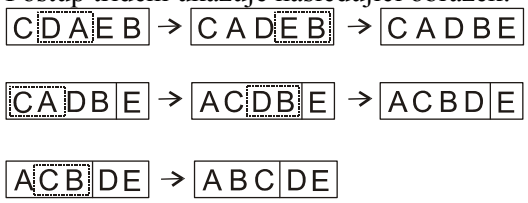
Z popisu bublinkového třídění je zřejmé, že tato třídící metoda se snadno implementuje. Zkuste si ji napsat v nějakém programovacím jazyce.

Řešení

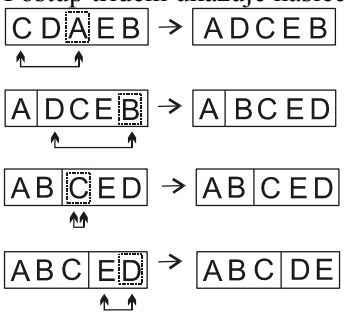
1. Postup třídění ukazuje následující obrázek:



2. Postup třídění ukazuje následující obrázek:



3. Postup třídění ukazuje následující obrázek:



Průvodce studiem

V naší zemi žije přibližně 10 miliónů obyvatel. Zkuste si spočítat, jak dlouho by doposud popsanými algoritmy trvalo setřídění jejich jmen podle abecedy, kdybychom

srovnání dvou jmen udělali vždy za 1 mikrosekundu.Určitě zjistíte, že je zapotřebí se zabývat i výkonnějšími algoritmy.

3.3 Účinnější metody vnitřního třídění

3.3.1 Shellovo třídění

Studijní cíle: Tato část vysvětluje princip Shellova třídění a poskytuje podrobný popis algoritmu tak, aby ho studující byl schopen případně implementovat.

Klíčová slova: Shellovo třídění.

Potřebný čas: 1 hodina

Shellovo třídění je založeno na třídění přímým vkládáním.

Shellovo třídění je třídění vkládáním. Vychází z již probrané metody třídění přímým vkládáním. Připomeňme, že průměrná složitost operací této metody je přibližně (po zanedbání lineárního a konstantního členu) $\frac{n^2}{4}$.

Aplikujme nyní třídění přímým vkládáním ne na celou posloupnost s n prvky, ale na h jejích podposloupností, kde h>1. Tyto podposloupnosti sestavíme tak, že budeme postupně vybírat prvky, jež jsou od sebe vzdáleny o h prvků:

1. podposloupnost: $a_1, a_{h+1}, a_{2*h+1}, \ldots$

2. podposloupnost: $a_2, a_{h+2}, a_{2*h+2}, \ldots$

….

h-tá podposloupnost: $a_h, a_{2*h}, a_{3*h}, \ldots$

Každá z těchto podposloupností místo n prvků má řádově jen $\frac{n}{h}$ prvků. Složitost operací je u každé z těchto podposloupností $\frac{\left(\frac{n}{h}\right)^2}{4} = \frac{n^2}{4h^2}$.

Podposloupností je h, složitost operací pro všech h podposloupností bude

$$h \times \frac{n^2}{4h^2} = \frac{n^2}{4h}$$

Rozdělením tříděné posloupnosti na více dílčích posloupností se dosáhne nižší složitosti.

To je viditelně méně, než když se přímé třídění provádí jen na jedné posloupnosti se všemi prvky najednou. Tříděním samostatných podposloupností se přirozeně nedosáhne setřídění celé posloupnosti. Prvky tak vesměs nebudou na své cílové pozici. Ale dostanou se tímto do určité blízkosti ke své cílové pozici. Jak blízko budou, bude záviset na hodnotě h. Čím bude h menší, tím blíže budou u své cílové pozice. Na druhé straně ale, čím h bude větší, tím příznivější bude složitost třídění všech podposloupností. Tento rozpor Shellovo třídění řeší tak, že dělení na více podposloupností nedělá jen jednou, ale vícekrát, přičemž postupně přitom snižuje velikost hodnoty h. Shellovo třídění tedy vychází z určité stanovené posloupnosti hodnot h

$$h_q, h_{q-1}, \ldots, h_2, h_1$$

která je klesající a poslední její hodnota je 1:

$$h_q > h_{q-1} > \ldots > h_2 > h_1 \qquad h_1 = 1$$

První z hodnot *h* je největší. Třídění pro ni proběhne rychle, ale přiblížení k cílové pozici bude hrubé. Druhá hodnota *h* už je menší. Zde už bude méně podposloupností a bude tedy každá už obsahovat více prvků.

Připomeňme, že při třídění přímým vkládáním se pozice vložení v setříděné části hledá tak, že setříděnou část procházíme odzadu. U prvního průchodu je hodnota *h* velká, čímž podposloupnosti jsou krátké a nalezení pozic vložení probíhá rychle. U dalších průchodů se délka podposloupnosti sice postupně zvyšuje, ale díky tomu, že bylo předtím vždy v předchozím průchodu provedeno určité přiblížení k cílové pozici, budou pozice vložení většinou poměrně blízko ke konci setříděné části, což znamená nižší počet srovnání a přesunů při vkládání do setříděné části. Tak se to opakuje až po poslední hodnotu *h*. Ta je ale 1, což znamená, že posledním průchodem je klasické třídění přímým vkládáním, jehož výsledkem je vždy setříděná posloupnost bez ohledu na to, jak blízko se prvky v předchozích krocích dostaly ke své cílové pozici. Tedy volba posloupnosti hodnot *h* má vliv jen na účinnost metody, nikoliv na výsledek třídění.

Příklad. Vezměme posloupnost

7 1 9 5 4 8 3 2

Posloupnost hodnot *h* zvolme

$$h_3 = 4, h_2 = 2, h_1 = 1$$

Začínáme první hodnotou $h_3 = 4$.

Vyznačme tučně první podposloupnost

7 1 9 5 **4** 8 3 2

A po jejím setřídění

4 1 9 5 **7** 8 3 2

Druhá podposloupnost před a po setřídění

4 **1** 9 5 7 **8** 3 2

4 **1** 9 5 7 **8** 3 2

Třetí podposloupnost před a po setřídění

4 1 **9** 5 7 8 **3** 2

4 1 **3** 5 7 8 **9** 2

A čtvrtá

4 1 3 **5** 7 8 9 **2**

4 1 3 **2** 7 8 9 **5**

Nyní vezmeme další hodnotu $h_2 = 2$.

První podposloupnost před a po setřídění

4 1 **3** 2 **7** 8 **9** 5

3 1 **4** 2 **7** 8 **9** 5

Druhá podposloupnost před a po setřídění

3 **1** 4 **2** 7 **8** 9 **5**

3 **1** 4 **2** 7 **5** 9 **8**

Na závěr proběhne běžné třídění přímým vkládáním.

Účinnost metody závisí na volbě posloupnosti hodnot

$$h_q, h_{q-1}, ..., h_2, h_1 \ .$$

Volba dělení na dílčí posloupnosti je kritická pro efektivnost metody.

Bude-li hodnot málo, budou skoky mezi nimi dosti velké a předchozí přiblížení bude relativně hrubé, čímž metoda bude méně efektivní. Bude-li jich naopak hodně, provede se zbytečně mnoho průchodů, čímž opět poklesne efektivita. Bylo zjištěno, že optimální posloupnost se vytvoří předpisem

$$h_1 = 1, \quad h_i = 2 \times h_{i-1} + 1 \quad pro \quad i > 1$$

což znamená posloupnost

…., 127, 63, 31, 15, 7, 3, 1

Zbývá otázka, kterou z těchto hodnot zvolit jako první. To závisí na počtu tříděných prvků *n*. Zvolíme největší z těchto hodnot takovou, aby v každé z podposloupností bylo co třídit, tedy aby obsahovala aspoň 2 prvky. Odtud pro první zvolenou hodnotu h_q dostáváme podmínku

$$h_q \leq \frac{n}{2} \ .$$

Budeme-li například třídit 100 prvků, použijeme posloupnost

31, 15, 7, 3, 1 .

Odvození časové složitosti Shellova třídění je obtížné. Uveďme proto jen výsledek $\Theta(n^{1.2})$.

Číslům *h* se také říká kroky Shellova třídění a samotná metoda bývá také nazývána tříděním vkládáním s ubývajícím krokem.

Kontrolní otázky

1. *Kdybychom v setříděné části hledali místo pro vložení dalšího prvku odpředu místo odzadu, mělo by to vliv na účinnost Shellova třídění?*
2. *Proč posloupnost kroků třídění volíme klesající?*
3. *Jaká je časová složitost Shellova třídění?*

Cvičení

1. Po jednotlivých krocích Shellovým tříděním setřiďte podle abecedy následujících sedm písmen. Zvolte přitom optimální posloupnost kroků.

C	D	F	A	H	E	B

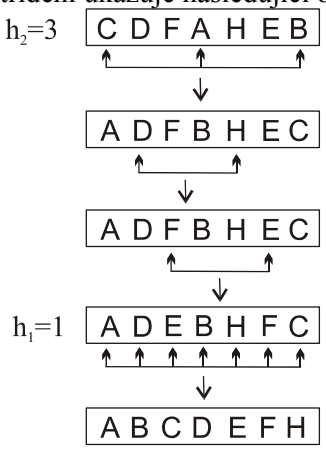
2. Máme setřídit 1000 prvků. Jaká je optimální posloupnost kroků?

Úkoly k textu

Zamyslete se nad tím, co kdybychom stejnou myšlenku rozdělení na více dílčích posloupností uplatnili u bublinkového třídění. Mělo by to nějaký zásadní vliv na složitost metody?

Řešení

1. Protože máme 7 prvků, optimální posloupnost kroků bude mít dva členy h_2=3 a h_1=1. Postup třídění ukazuje následující obrázek:



2. Máme-li třídit 1000 prvků, můžeme je rozdělit do nejvýše 500 dílčích posloupností, aby v každé byly aspoň dva prvky. Podíváme-li se na optimální posloupnost kroků, vidíme, že její členy jsou mocniny čísla 2 zmenšené o 1. Tedy hledáme největší číslo *k*, proto které platí:

$$500 \geq 2^k - 1$$

Zřejmě tomu vyhovuje *k*=8 a optimální posloupnost kroků je

255, 127, 63, 31, 15, 7, 3, 1

3.3.2 Rychlé třídění výměnou (Quicksort)

Studijní cíle: Tato část vysvětluje princip rychlého třídění výměnou a poskytuje podrobný popis algoritmu tak, aby ho studující byl schopen případně implementovat.

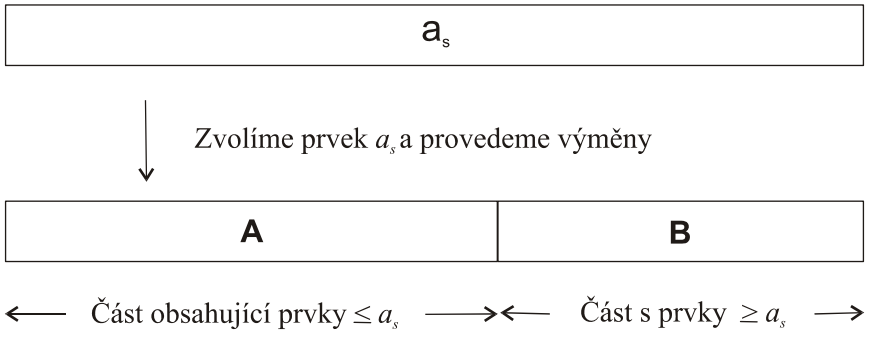
Klíčová slova: Quicksort.

Potřebný čas: 1 hodina 30 minut.

Budeme vycházet opět z předpokladu, že třídění probíhá v poli, přičemž každý prvek pole reprezentuje jeden tříděný prvek. Počet tříděných prvků je *n* a indexování pole je od 0, což je v dnešních programovacích jazycích obvyklé.

Princip metody spočívá v tom, že v poli zvolíme určitý prvek, označme ho a_s, a postupnými výměnami z tříděného pole vytvoříme dvě části. V první části (označme ji A) budou prvky, které nejsou větší než zvolený prvek a_s, a v druhé části (B) budou prvky, které nejsou menší než a_s.

Principem metody je provedení výměn tak, že pole se jimi rozdělí na dvě části, mezi nimiž už žádné výměny nebudou.



Je zřejmé, že v dalším pokračování třídění, tj. při provádění dalších výměn už stačí jen vyměňovat samostatně prvky obsažené v části A a samostatně prvky obsažené v části B. Tudíž postup, který jsme na začátku aplikovali na celé pole, nyní rekurzivně aplikujeme samostatně jen na část A a pak na část B. Postupným rekurzivním opakováním tohoto postupu se tříděné části stávají postupně stávají menší, až nakonec nastane stav, že obsahují jen jeden prvek, čímž to končí.

Po rozdělení pole na dvě části se stejný postup rekurzivně aplikuje na obě části.

Klíčovou záležitostí je volba prvku a_s. Optimální by bylo zvolit ho tak, aby vzniklé části A a B měly stejnou velikost (stejný počet prvků). To by znamenalo najít prvek, jehož hodnota je uprostřed hodnot všech prvků. Nalezení takového prvku je ale natolik časově náročné, že by se to celkově nevyplatilo. Proto se volí mnohem jednodušší způsob - vezme se prvek, který v poli je uprostřed.

V popisu algoritmu použijeme značení:

a – je jméno pole s tříděnými prvky

k,l – jsou indexy počátku a konce části pole, která je právě tříděna

r – je index prvku, který je uprostřed tříděné části

i,j – jsou průběžné indexy používané pro procházení polem ve směrech odpředu a odzadu

Popis algoritmu

1. Počáteční krok

Na počátku právě tříděnou částí bude celé pole, tj. nastavíme

$$k = 0 \qquad l = n - 1 \ .$$

2. Třídící krok

Najdeme index prvku, který je uprostřed tříděné části pole

$$s = \frac{k + l}{2}$$

Přesněji řečeno, pokud tříděná část má sudý počet prvků, pak je to index prvního ze dvou prvků, které jsou uprostřed. Prvek pole s tímto indexem a_s si uložíme do proměnné, kterou si označíme x.

Průběžné indexy nastavíme na začátek a konec právě tříděné části

$$i = k \qquad j = l$$

Nyní procházíme tříděnou část směrem dozadu, zvětšujeme index i, tak dlouho, až najdeme prvek, pro který platí

$$a_i \geq x \ .$$

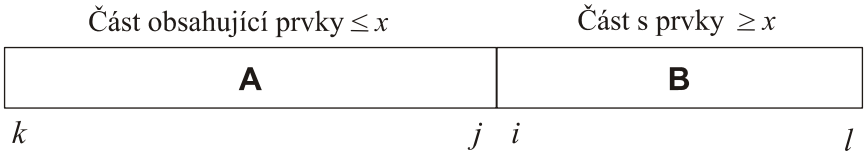
Následně procházíme tříděnou část směrem dopředu, snižujeme index j, tak dlouho, až najdeme prvek, pro který platí

$$a_j \leq x \ .$$

Prvky a_i a a_j mezi sebou vyměníme a následně zvýšíme hodnotu indexu i a snížíme hodnotu indexu j

$$i = i + 1 \qquad j = j - 1$$

Proces hledání a výměn provádíme tak dlouho, dokud nenastane $i>j$. V tomto okamžiku je buďto $i=j+1$, pak pole je rozděleno na dvě části:



Příklad. Tříděné pole nechť obsahuje

3 4 1

Na začátku je $k = 0$, $l = 2$.

Vypočítáme

$$s = \frac{0+2}{2} = 1 \quad , \quad x = a_1 = 4$$

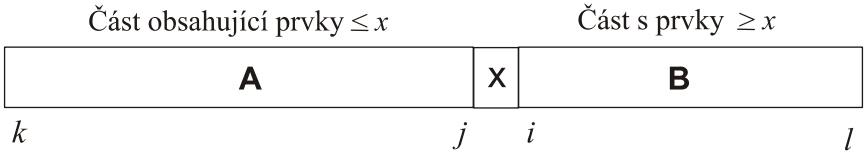
a položíme $i = 0$, $j = 2$.

Po hledání zleva je $i = 1$, neboť je $a_1 \geq x$ (4≥4), a po hledání zprava je j=2 neboť je $a_2 \leq x$ (1≤4). Po výměně prvků a_1 a a_2 a přičtení 1 k indexu i a odečtení je 1 od indexu j je rozdělení pole

3 1 | 4 ,

neboť hodnoty indexů jsou nyní $i = 2, j = 1$.

Druhá z možností je, že bude platit $i=j+2$. Pak z pole se vydělí opět dvě části, mezi nimiž je ale ještě jeden prvek, jehož hodnota je stejná, jako je v proměnné x:



Příklad. Tříděné pole nechť obsahuje

5 3 4 1

Na začátku je $k = 0$, $l = 3$.

Vypočítáme

$$s = \frac{0+3}{2} = 1 \quad , \quad x = a_1 = 3$$

a položíme $i = 0$, $j = 3$.

Po hledání zleva je $i = 0$, neboť $a_0 \geq x$ (5≥3), a pohledání zprava je j=3, neboť $a_3 \leq x$ (1≤3). Po výměně bude pole

1 3 4 5

a nové hodnoty indexů $i = 1, j = 2$.

Po pokračování hledání zleva bude $i = 1$, neboť $a_1 \geq x$ (3≥3), a po pokračování hledání zprava bude j=2, neboť $a_1 \leq x$ (3≤3). Po výměně (prvek a_1 se vymění sám se sebou) bude pole

1 3 4 5

a nové hodnoty indexů $i = 2, j = 1$. Tudíž výsledné rozdělení pole je

1 | 3 | 4 5 .

Má-li část *A* více než jeden prvek, pak rekurzivně provedeme třídící krok na této části pole – její počáteční a koncový index je dán současnými hodnotami proměnných *k,j*.

Má-li část *B* více než jeden prvek, pak rekurzivně provedeme třídící krok na této části pole – její počáteční a koncový index je dán současnými hodnotami proměnných *i,l*.

Příklad. Vezměme posloupnost

7 1 9 5 4 8 3 2

Na začátku $i = 0, j = 7$, střední prvek je $x = 5$.

Po hledání $i = 0, j = 7$

7 1 9 5 4 8 3 **2**

a výměna

2 1 9 5 4 8 3 **7**

Nové indexy $i = 1, j = 6$. Pod dalším hledání $i = 2, j = 6$

2 1 **9** 5 4 8 **3** 7

a výměna

2 1 **3** 5 4 8 **9** 7

Nové indexy $i = 2, j = 5$. Pod dalším hledání $i = 3, j = 4$

2 1 3 **5 4** 8 9 7

a výměna

2 1 3 **4 5** 8 9 7

Nové indexy $i = 4, j = 3$ - třídící krok končí. Rozdělení pole je

2 1 3 4 | 5 8 9 7

Nyní vezmeme část *A*:

2 1 3 4

Na začátku $i = 0, j = 3$, střední prvek je $x = 1$.

Po hledání $i = 0, j = 1$

2 1 3 4

a výměna

1 2 3 4

Nové indexy $i = 1, j = 0$ - třídící krok končí. Rozdělení pole je

1 | 2 3 4

Část *A* už má jeden prvek, vezmeme část *B*:

2 3 4

Na začátku $i = 1, j = 3$, střední prvek je $x = 3$.

Po hledání $i = 2, j = 2$

2 **3** 4

a výměna (sama se sebou)

2 **3** 4

Nové indexy $i = 3, j = 1$ - třídící krok končí. Rozdělení pole je

2 | 3 | 4

čímž je tato část dokončena.

Zbývá ještě část *B* z prvního třídícího kroku:

5 8 9 7

Na začátku $i = 4$, $j = 7$, střední prvek je $x = 8$. Atd.

Složitost metody

Stanovení složitosti metody je poněkud problematičtější. Ta závisí na tom, v jakém poměru se tříděná část rozdělí na nové části *A* a *B*. Nejhorší případ nastane, když jedna z těchto částí obsahuje jen jeden prvek a ta druhá zbývající prvky. Pak by následující třídící krok proběhl vždy jen pro tu větší část a ta by měla vždy o jeden prvek méně, než tomu bylo v předchozím kroku. Tedy třídící kroky by proběhly postupně pro části s počty prvků n, n-1, …, 2. Počet srovnání v každém třídícím kroku bude záviset jednak na počtu prvků v procházené části a také určitým způsobem na tom, kolik proběhne výměn (neboť po každé výměně se změní oba indexy). Počet srovnání je maximálně o 1 větší, než je počet prvků v procházené části. Tento největší počet srovnání nastane v případě, kdy procházená část už je setříděná. Nechť například procházená část je

0 1 3 4 5 7 8 9 .

Její střední prvek je $x = a_3 = 4$.

Budou-li na začátku indexy $i = 0$, $j = 7$, proběhnou zleva 4 srovnání, než index i skončí na hodnotě i=3, neboť $a_3 \geq x$. Zprava proběhne 5 srovnání, než index j skončí na hodnotě j=3, neboť $a_3 \leq x$. Tedy celkový počet srovnání je 9, což je o 1 více než je počet prvků v procházené části, neboť v ní je 8 prvků.

Maximální celkový počet srovnání $= (n+1) + n + \ldots + 3 = (n+1) + n + \ldots + 3 + 2 + 1 - 3 =$

$$= \frac{(n+1)(n+2)}{2} - 3$$

Pro počet výměn nám stačí skutečnost, že jich je nejvýše tolik, kolik je srovnání. Celkově je z toho vidět, že složitost operace srovnání a tím i celé metody je v tomto nejhorším případě kvadratická.

Naopak optimální případ nastane, když procházená část se vždy rozdělí na dvě stejné části (při lichém počtu prvků na dvě části lišící se o jeden prvek). Možnost, že dělení také může být na dvě části a střední prvek, pro jednoduchost opomineme. Spočítáme, kolik nyní bude zapotřebí třídících průchodů. Sestavíme pro ně následující tabulku

Třídící průchod	Délka částí
1	n
2	$\frac{n}{2}$
3	$\frac{n}{2^2}$
….	
h	$\frac{n}{2^{h-1}}$

Každá vytvořená část vstupuje do dalšího třídění, má-li ještě aspoň dva prvky. Pro poslední třídící průchod z toho dostáváme vztah

$$\frac{n}{2^{h-1}} = 2 \text{ odtud } 2^h = n \text{ a dále } h = \log_2(n)$$

Tedy počet třídících chodů logaritmicky závisí na počtu tříděných prvků. Zbývá stanovit složitost operace srovnání v jednom třídícím průchodu. Ta je závislá na tom, na kolik částí je tříděné pole rozděleno. Máme-li v daném třídícím chodu *k* částí, pak v každé z nich bude $\frac{n}{k}$ prvků. Počet srovnání v každé části je nejvýše o 1 větší než je počet prvků v ní, odtud

Počet srovnání v jednom třídícím průchodu $= k \times \left(\frac{n}{k} + 1 \right)$.

V prvním třídícím průchodu chodu je počet částí 1, v posledním, kdy části mají délku 2, je počet $\frac{n}{2}$. Dosazením zjistíme, že počet srovnání v jednom třídím průchodu se pohybuje od *n*+1 v prvním třídícím průchodu až po $\frac{3n}{2}$ v posledním průchodu. Tedy ve všech průchodech je složitost lineární. Když vynásobíme složitost jednoho průchodu počtem průchodů, dostaneme celkovou složitost:

$$O(n \times \log(n)) \ .$$

Nyní je otázka, jak tomu bude v běžném případě. Ukazuje se, že míra složitosti v běžném případě je stejná jako v optimálním. Pesimistická kvadratická složitost nejhoršího případu se u běžných dat prakticky nevyskytuje.

*Složitost metody je typicky Θ(n*log(n)).*

Závěr: Složitost metody je typicky $O(n \times \log(n))$.

Popsaná metoda je známa zejména pod svým původním anglickým názvem Quicksort (rychlé třídění).

Průvodce studiem

Algoritmus pro Quicksort vznikl v roce 1962. Tři roky po vzniku Shellova třídění, které pochází z roku 1959.

1. *Proč jako referenční prvek volíme prostřední prvek v poli, ačkoliv optimální by bylo zvolit prvek, jehož hodnota je uprostřed hodnot všech prvků?*
2. *Jaké mohou nastat případy rozdělení pole na dvě části, v popisu označené A a B, tj. část,i mezi kterými už nebudou žádné výměny?*
3. *Jaká je časová složitost třídění Quicksort?*

Cvičení

1. Metodou Quicksort setřiďte podle abecedy následujících sedm písmen.

C	D	F	A	H	E	B

2. Jak bude třídění probíhat pro následující opačně uspořádanou posloupnost pěti písmen:

E	D	C	B	A

Úkoly k textu

V popis metody jsme uvedli, že při výběru referenčního prvku by bylo optimální vzít prvek, jehož hodnota je uprostřed všech hodnot v procházené části pole. Místo toho ale bereme prvek, který leží uprostřed. Tedy máme-li posloupnost prvků

2 5 8 4 9

pak prvek uprostřed je 8, což následně vede na rozdělení na části

2 5 4 8 - 9 .

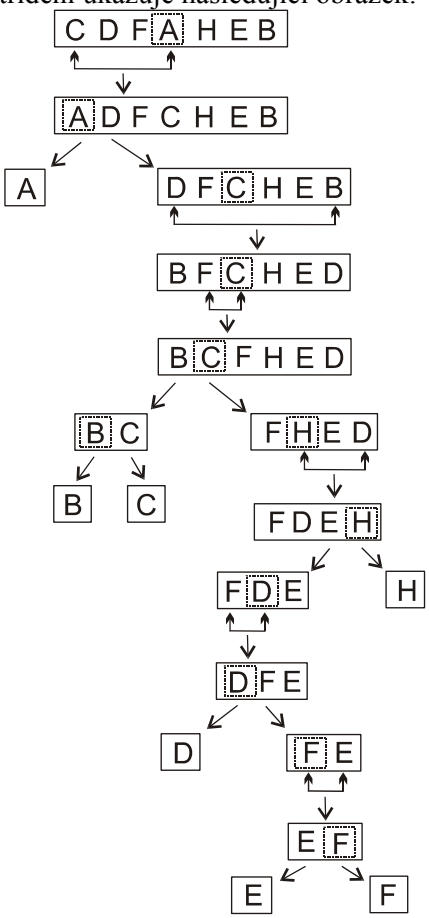
Ale kdybychom vzali prvek s hodnotou uprostřed, což je 5, bylo by rozdělení na části mnohem příznivější

2 4 5 - 8 9 .

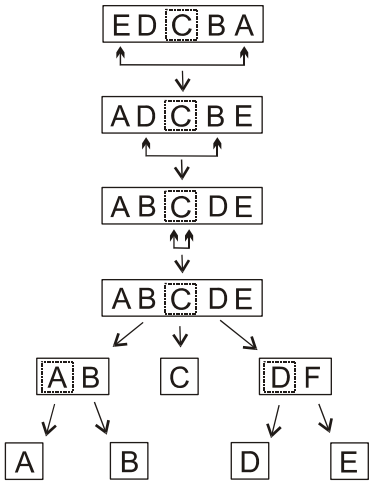
Jako důvod jsme uvedli. že nalezení prvku s hodnotu uprostřed je značně časově náročné. Zkuste se zamyslet, jak by asi vypadal algoritmus pro nalezení prvku s prostřední hdonotou a jakou by měl asi časovou složitost. Konkrétně bychom potřebovali, aby měl lineární složitost. Neboť pak bychom vzhledem k tomu, že dělení v tomto případě vede na dvě přibližně stejné části a tedy na logaritmickou závislost počtu třídících kroků, měli zaručenou celkovou složitost rovnu součinu lineární a logaritmické, což je $\Theta(n*log(n))$.

Řešení

1. Postup třídění ukazuje následující obrázek:



2. Postup třídění ukazuje následující obrázek:



3.3.3 Třídění použitím haldy

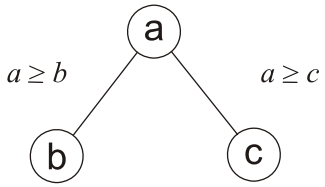
Studijní cíle: Tato část vysvětluje princip třídění haldou a poskytuje podrobný popis metody tak, aby ji studující byl schopen případně implementovat.

Klíčová slova: Halda, binární strom.

Potřebný čas: 2 hodiny

Halda je specifický binární strom.

Poslední popisovaná metoda vnitřního třídění používá k třídění haldu. Halda je určitý typ binárního stromu. V každém jeho uzlu je jeden tříděný prvek. Dále mezi prvkem v uzlu a prvky uloženými v jeho následnících (má-li nějaké) platí vztah, že prvek v uzlu má z nich největší hodnotu.



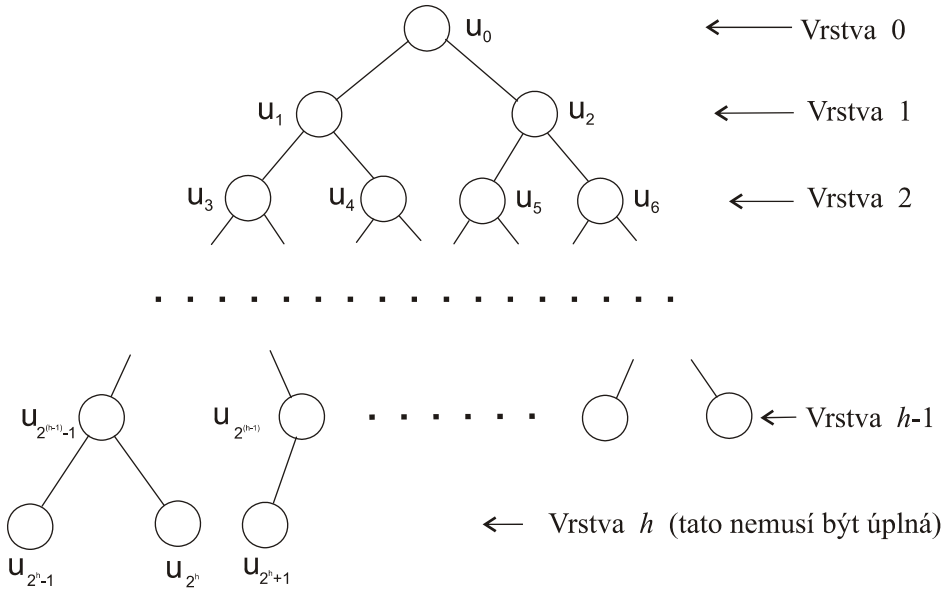
Popis algoritmu

Proces třídění má dvě fáze – vytvoření haldy a vlastní třídění.

Fáze 1 – vytvoření haldy

Haldu sestavujeme jako vyvážený strom.

V první fázi vytvoříme haldu. Halda je binární strom a jak již bylo uvedeno v kapitole popisující stromovou strukturu, optimální je vyvážený binární strom. V něm jsou všechny vrstvy zcela zaplněné až na poslední vrstvu, která jediná může být neúplná. Připomeňme, že vrstvou jsme nazvali množinu uzlů, které mají od kořene stejnou vzdálenost, tedy leží na stejné vodorovné úrovni. Vzhledem k tomu, že pro implementaci haldy, jak si ukážeme na konci, používáme pole a pole je běžně indexováno od 0, budeme i uzly haldy indexovat od 0.



Všimněme si specifické vlastnosti indexů jednotlivých uzlů ve vyváženém binárním stromu:

- Uzly ve vrstvě *k* mají indexy v rozmezí $2^k - 1 \ldots 2^{(k+1)}$
- Následníci (existují-li) uzlu s indexem *i* mají indexy 2∗*i*+1 a 2∗*i*+2, tj. následníci uzlu u_i jsou uzly u_{2*i+1} a u_{2*i+2}

Nejdříve si z počtu tříděných prvků *n* vypočítáme výšku haldy, označme ji *h*:

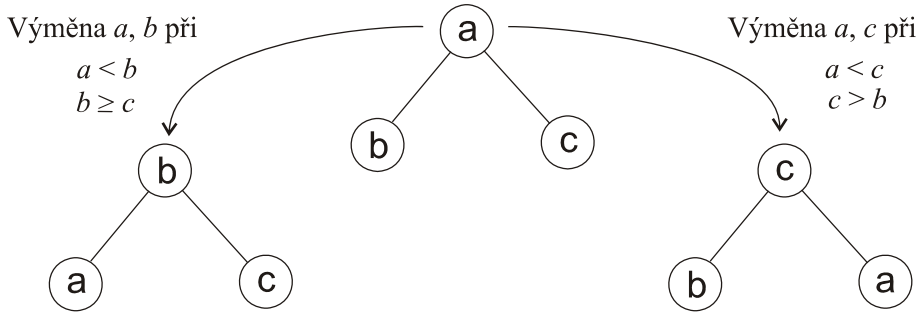
$$h = \log_2(n)$$

Sestavíme si vyvážený binární strom této výšky s *n* uzly a zaplníme ho tříděnými prvky, tj. do každého uzlu dáme jeden tříděný prvek.

Následně budeme odspodu procházet nelistové uzly a budeme ověřovat, zda splňují podmínku haldy vzhledem ke svým následníkům. Tedy budeme postupně brát následující uzly

$$u_{\frac{n}{2}-1}, u_{\frac{n}{2}-2}, \ \ldots, u_1, u_0$$

v tomto pořadí a u každého z těchto uzlů ověříme, zda prvek v něm uložený není menší než prvek v některém z jeho následníků. Pokud ano, uděláme výměnu dle následujícího schématu:



Jestliže došlo k výměně, je nyní v příslušném následníku jiný prvek, čímž může u něho dojít k narušení podmínky haldy vzhledem k prvkům v jeho následnících. Musíme tedy obdobné srovnání (a případnou výměnu) nyní provést pro tohoto následníka. Takto budeme postupovat směrem dolů tak dlouho, dokud nedojdeme buďto k uzlu, u kterého výměna není nutná, anebo k uzlu, který už žádné následníky nemá (listu).

Fáze 2 – vlastní třídění

Třídění probíhá přesunem prvků z kořenu do listového uzlu haldy.

Z vlastností haldy plyne, že v jejím kořenu je největší prvek. Ten vyměníme s prvkem v posledním listu haldy. O tento list následně haldu zkrátíme a pro nový prvek v kořenu ověříme, zda splňuje vlastnost haldy vzhledem ke svým následníkům. Pokud ne, vyměňujeme prvky mezi uzly a jejich následníky tak dlouho, až všechny splňují vlastnost haldy. Jde o stejný postup, jakým jsme ve fázi vytváření haldy prováděli výměny prvků mezi nelistovými uzly a jejich následníky.

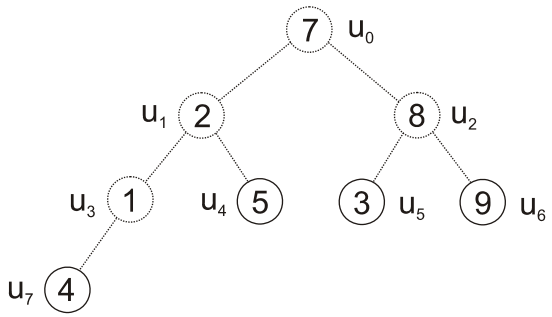
Je zřejmé, že každým krokem se halda zkrátí o jeden uzel. Až nakonec obsahuje jen jeden uzel, čímž proces třídění končí. Setříděné posloupnost je na konci v uzlech binárního stromu, který jsme začátku vytvořili a do kterého jsme uložili tříděné prvky. Dostaneme ji tak, že strom procházíme shora po jednotlivých vrstvách směrem dolů, tj. tvoří ji uzly stromu v pořadí

$$u_0, u_1, \ ..., u_{n-2}, u_{n-1} \quad .$$

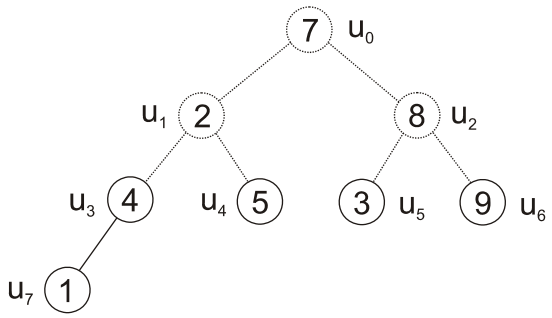
Příklad. Nechť tříděné hodnoty jsou

7 2 8 1 5 3 9 4

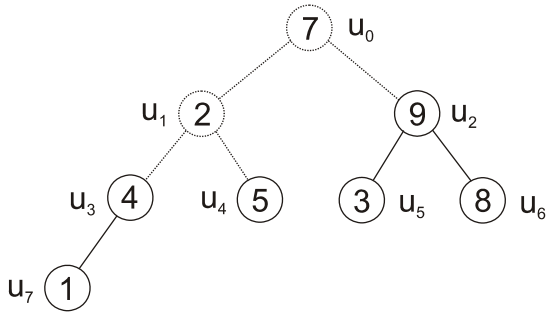
Jejich počet je n=8. Zřejmě $\log_2(8)=3$, tedy výška stromu je 3. Strom vytvoříme a do jeho uzlů dáme tříděné prvky:



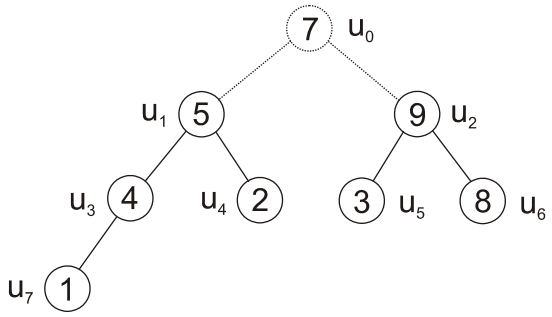
Vezmeme poslední z nelistových uzlů u_3 a ověříme velikost hodnoty uložené v tomto uzlu s hodnotou v jeho následníku u_7. Je vidět, že hodnota v následníku je menší a je tedy potřeba výměna hodnot.



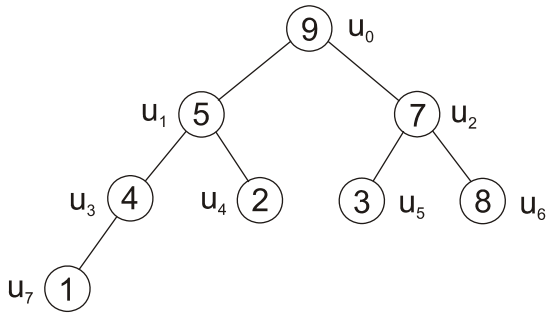
Vezmeme další nelistový uzel u_2. U něho je nutná výměna hodnot mezi tímto uzlem a jeho pravým následníkem u_6.



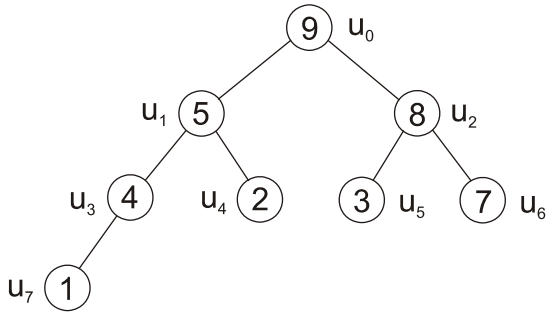
Další uvažovaný uzel je u_1. Zde je nutná výměna mezi tímto uzlem a jeho pravým následníkem u_4.



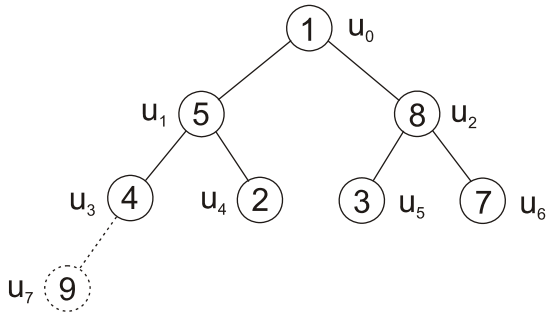
Zbývá poslední doposud neuvažovaný uzel u_0. U něho je zapotřebí vyměnit hodnotu v něm s hodnotou uloženou v jeho pravém následníku u_2.



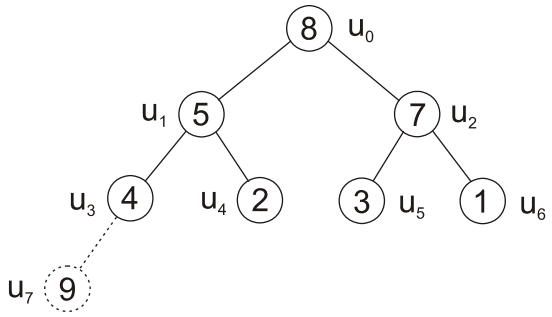
Je zřejmé, že po této výměně nyní hodnota v uzlu u_2 nesplňuje vlastnost haldy a je ji nutno vyměnit s hodnotou v pravém následníku u_6.



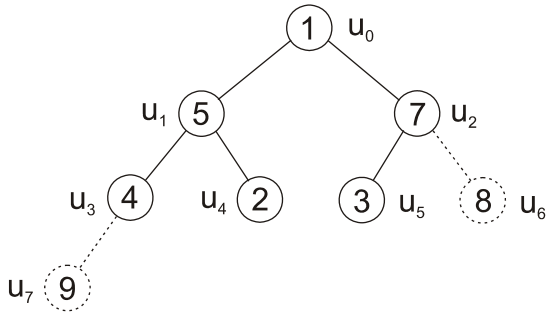
Nyní můžeme začít třídění. Hodnotu v kořenu vymění s hodnotou v posledním listu u_7 a haldu o tento list zkrátíme.



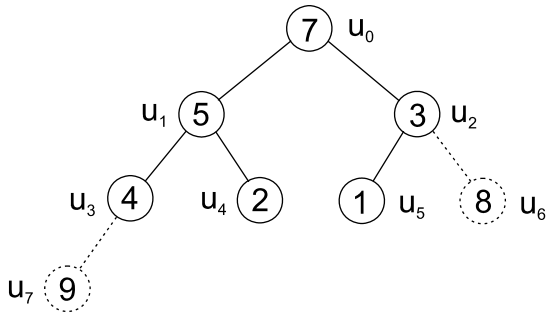
Nyní je nutno vyměnit hodnoty postupně mezi uzly mezi u_0 a u_2 a uzly u_2 a u_6.



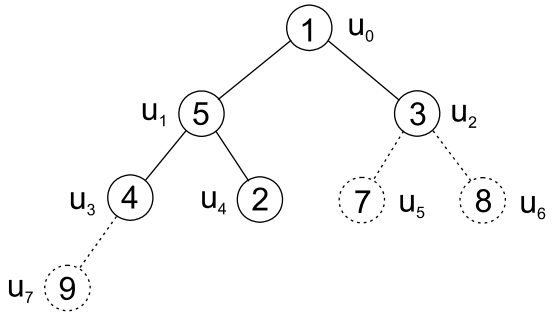
Hodnotu v kořenu vymění s hodnotou v posledním listu u_6 haldy a haldu o tento list zkrátíme.



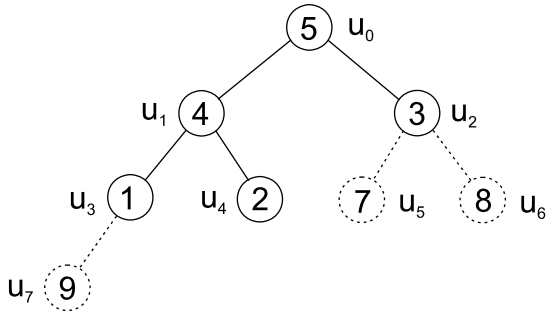
Nyní je nutno vyměnit hodnoty mezi uzly u_0 a u_2 a uzly u_2 a u_5.



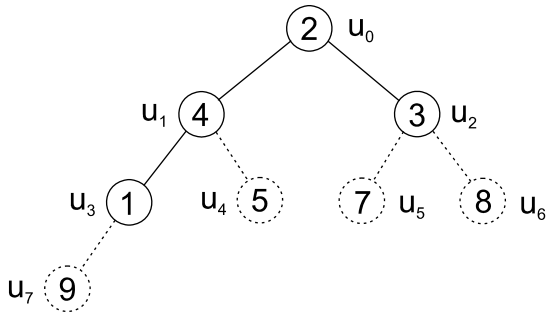
Hodnotu v kořenu vymění s hodnotou v posledním listu u_5 haldy a haldu o tento list zkrátíme.



Následuje výměna hodnot postupně mezi uzly mezi u_0 a u_1 a uzly u_1 a u_3.



Hodnotu v kořenu vymění s hodnotou v posledním listu u_4 haldy a haldu o tento list zkrátíme.



Atd.

Složitost metody

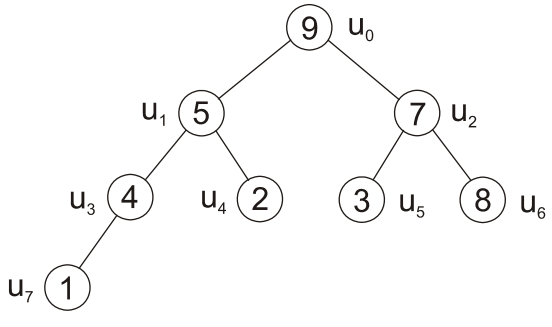
Vezměme nejprve složitost vytvoření haldy. Při něm procházíme jednotlivé jeho nelistové uzly, srovnáváme prvky v něm uložené s následníky a případně provádíme přesuny. Srovnání je v každém uzlu nutné provést dvě - se dvěma následníky (má-li uzel oba následníky). Následně proběhne případný přesun. Vezměme ten nejhorší případ, kdy srovnání a přesuny proběhnou od nejhornějšího uzlu (kořene) až po ten nejspodnější nelistový uzel. Jde celkem o $3*h$ operací (3 operace v každém uzlu: 2 srovnání + 1 přesun), kde h je výška stromu. Výška binárního vyváženého stromu logaritmicky závisí na počtu uzlů (prvků). Tedy pro každý probíraný nelistový jeden uzel má v nejhorším případě (a i v průměrném případě) logaritmickou složitost. Když ji vynásobíme počtem $\frac{n}{2}$ uvažovaných nelistových uzlů, dostaneme složitost vytvoření haldy $\Theta(n*log(n))$.

Složitost třídění haldou je $\Theta(n*log(n))$.

Zbývá stanovit složitost fáze vlastního třídění. Ta je hlavně určena složitostí operací srovnání a případných výměn, které následují po vzájemné výměně prvků mezi kořenem a posledním listem ve stávající haldě a zkrácením haldy o tento list. Opět to vyžaduje 2 srovnání a 1 přesun na každý nelistový uzel (nebo 1 srovnání, pokud uzel má jen jednoho následníka). Tedy maximálně $3*h$ operací (tento počet typicky klesá, jak se halda postupně zkracuje). Opět logaritmická složitost pro každý nový prvek v kořenu haldy. Vynásobíme-li ji celkovým počtem prvků, dostáváme zase složitost $\Theta(n*log(n))$.

Závěr: Složitost třídění použitím haldy je $\Theta(n*log(n))$.

I když tato třídící metoda je založena na stromové struktuře, při implementaci ji nemusíme dynamicky vytvářet pomocí uzlů, ukazatelů atd. Jak už jsem ukázali, následníci uzlu s indexem i mají indexy $2*i+1$ (levý následník) a $2*i+2$ (pravý následník). Strom se tímto dá snadno realizovat pomocí pole, když jeho prvky dáme po řádcích do pole.



9	5	7	4	2	3	8	1
u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7

Kontrolní otázky

1. *Jakým způsobem sestavíme z posloupnosti prvků haldu?*
2. *Proč haldu sestavujeme jako vyvážený strom? Kdyby nebyl vyvážený, mělo by to nějaký vliv na výsledné setřídění prvků?*
3. *Jaká je časová složitost třídění haldou?*

Cvičení

1. Tříděním haldou setřiďte podle abecedy následujících sedm písmen.

C	D	F	A	H	E	B

2. Jak bude probíhat třídění haldou pro následující obráceně uspořádanou posloupnost pěti písmen:

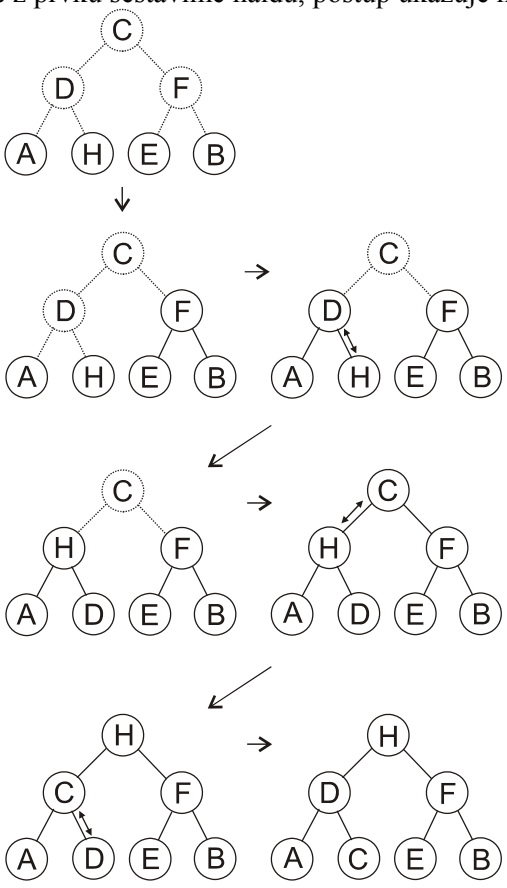
E	D	C	B	A

Úkoly k textu

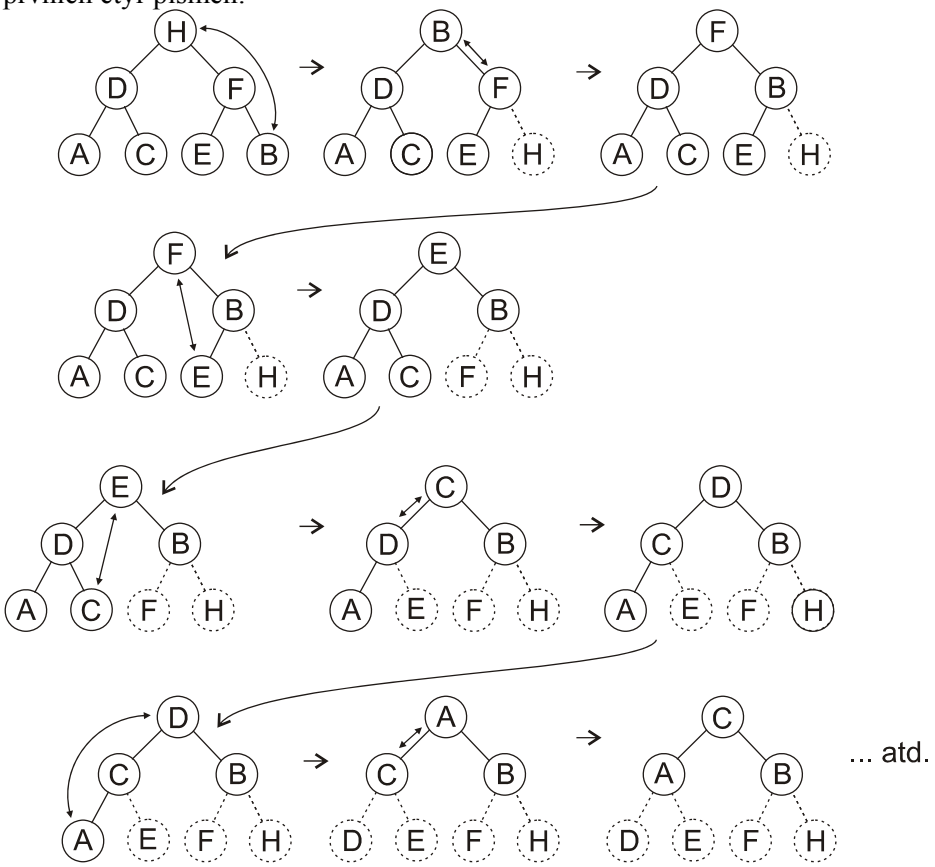
Občas je zapotřebí třídit v opačném pořadí, tj. od největšího prvku k nejmenšímu. Zamyslete se, jak by se to realizovalo při třídění haldou.

Řešení

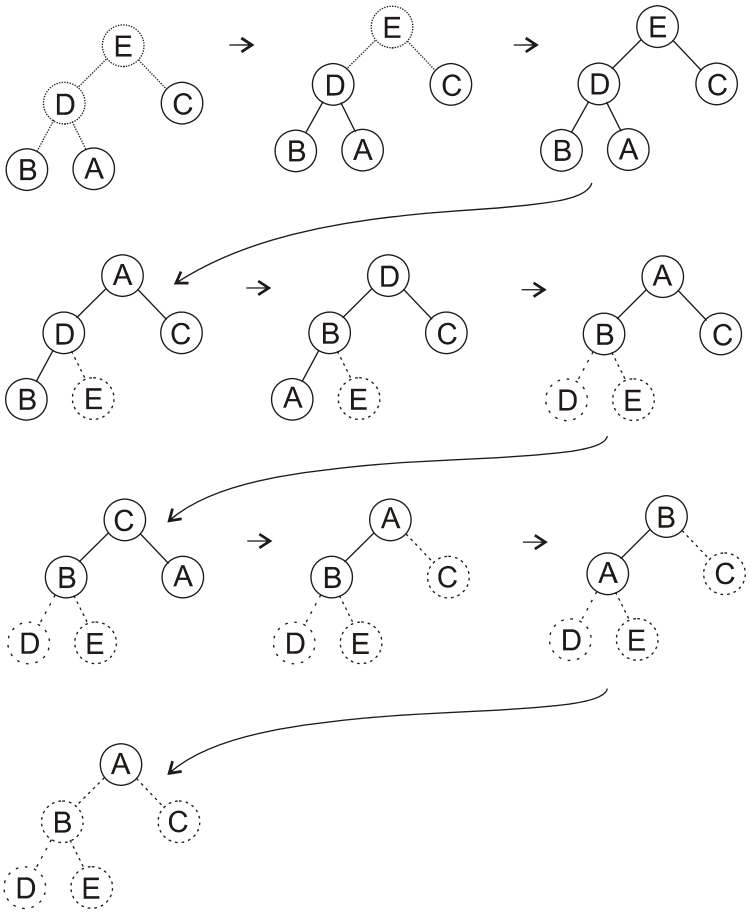
1. Nejprve z prvků sestavíme haldu, postup ukazuje následující obrázek:



Následně výměnami prvků mezi kořenem a posledním listem haldy a zkracováním haldy provedeme třídění. Následující obrázek ukazuje první čtyři kroky, tj. setřídění prvních čtyř písmen:



2. Postup sestavení haldy ukazuje následující obrázek:



3.3.4 Srovnání metod vnitřního třídění

Probrali jsme celkem 6 metod vnitřního třídění. První 3 byly přímé, jednodušší metody. Jejich složitost byla u všech stejná $\Theta(n^2)$. Protože je to citelně vyšší složitost než u dokonalejších metod, použijeme je jen v případech, kdy počet tříděných prvků není tak velký. Z odhadů, který jsme učinili v první kapitole o složitosti, že dá říci, že přímé metody vyhovují do řádově tisíce tříděných prvků. Implementačně nejjednodušší z nich je bublinkové třídění. Celý program pro tuto metody se skládá ze dvou vnořených cyklů. Nejrychlejší z nich je třídění přímým vkládáním. Plyne to ze skutečnosti, že tato metoda ve srovnání se zbývajícími dvěma metodami má méně operací srovnání, což má vliv na čas tříděný.

Nejpoužívanější metodou je Quicksort.

Ze zbývajících tří metod má nejhorší složitost Shellovo třídění $(\Theta(n^{1,2}))$. Věcně není žádný důvod, proč bychom ho měli zvolit, neboť máme další dvě metody, jež mají lepší složitost. Metoda Quicksort má očekávanou složitost $\Theta(n*log(n))$ a třídění haldou ji má dokonce zaručenou. Dá se i dokázat, že nelze sestavit třídící metodu, která by měla lepší složitost než $\Theta(n*log(n))$. Lepší složitost mají jen určité specifické metody jako je adresní třídění, které jsou ale použitelné jen ve značně speciálních případech. Kdybychom si nyní měli vybrat podle toho, jak obtížná je jejich implementace, je zřejmé, že bychom zvolili metodu Quicksort. Ta se naprogramuje poměrně snadno. Je to jedna rekurzivní procedura se dvěma formálními parametry – počátečním a koncovým indexem tříděné části pole. V praxi se také tato metoda nejčastěji používá. Přitom je nejen implementačně jednodušší, ale běžně má i nižší čas třídění než má třídění haldou. Příčina, proč třídění haldou trvá déle, tkví v jeho poměrně komplikovaném algoritmu.

3.4 Vnější třídění

Studijní cíle: Tato část vysvětluje princip vnějšího třídění a poskytuje podrobný popis metody tak, aby ji studující byl schopen případně implementovat.

Klíčová slova: Běh, zatřiďování, polyfázové třídění.

Potřebný čas: 1 hodina 30 minut

Vnější třídění je určeno pro velké objemy dat.

Na rozdíl od vnitřního třídění, kdy všechny tříděné prvky jsou ve vnitřní paměti, při vnějším třídění jsou tříděné prvky uloženy v souborech na vnějších pamětích (pevném disku). Třídění probíhá tak, že tříděné prvky jsou ze souborů po malých počtech přenášeny do vnitřní paměti, v ní jsou zatřiďovány a pak opět ukládány do souborů. Protože operace čtení ze souborů a zápis do souborů jsou výrazně pomalejší, než když přesuny prvků probíhají výlučně ve vnitřní paměti, je vnější třídění obecně pomalejší než vnitřní třídění a použijeme ho jen v případě, kdy vnitřní třídění nelze použít, protože tříděných prvků je tak mnoho, že se všechny do paměti nevejdou.

3.4.1 Třídění se stejným počtem vstupních a výstupních souborů

Třídění se stejným počtem vstupních a výstupních souborů je v podstatě nejjednodušší varianta vnějšího třídění. Označme si celkový počet souborů použitých při třídění r. Protože vstupních i výstupních souborů je stejný počet, musí to být sudé číslo. Minimální počet vstupních souborů, aby mohla probíhat operace zatřiďování, je 2, čímž celkový nejmenší možný počet souborů je 4. Pro jednodušší zápis si zaveďme i samostatné označení pro počet vstupní souborů, označme si ho v ($v = \frac{r}{2}$).

Vnější třídění je založeno na operaci zatřiďování. Při ní je z více setříděných sekvencí vytvářena jedna delší setříděná sekvence. Setříděným sekvencím budeme říkat běhy. Na začátku třídění běhy mají délku 1, jsou tvořeny jedním prvkem. Opakovaným procesem zatřiďování jsou z nich vytvářeny stále delší a delší běhy, až se dosáhne stavu, že je vytvořen běh tak dlouhý, že obsahuje všechny tříděné prvky. Tím třídění končí.

Popis algoritmu

Proces třídění má dvě fáze – rozdělování a zatřiďování.

Fáze 1 – rozdělování

První fáze je poměrně rychlá. Jejím cílem je pravidelně rozdělit tříděné prvky do v souborů, abychom mohli zahájit vlastní třídící proces. Vezmeme ze stanoveného počtu r souborů polovinu (v) a otevřeme je pro výstup (zápis). Do nich pravidelně rozděluje tříděné prvky. První prvek zapíšeme do prvního souboru, druhý prvek do druhého souboru, …, až v-tý prvek do v-tého souboru, následující prvek opět do prvního souboru atd.

Tedy tříděné prvky

$$a_1, a_2, a_3, ..., a_n$$

rozdělíme do souborů následovně

1. soubor: $a_1, a_{v+1}, a_{2*v+1}, ...$

2. soubor: $a_2, a_{v+2}, a_{2*v+2}, ...$

….

v-tý soubor: $a_v, a_{2*v}, a_{3*v}, ...$

Po dokončení rozdělování soubory výstupní soubory uzavřeme a následně je otevřeme jako vstupní (pro čtení) a zbývající polovinu souborů otevřeme jako výstupní. Délku běhů nastavíme na 1.

Fáze 2 - zatřiďování

Ze všech *v* vstupních souborů načteme jejich první běh, zatříděním z něho vytvoříme jeden delší výstupní běh a ten uložíme do prvního výstupního souboru. Pak ze vstupních souborů načteme druhý běh, z něho opět vytvoříme výstupní běh a uložíme ho do druhého výstupního souboru. A tak pokračujeme až *v*-tý vytvořený výstupní běh uložíme do *v*-tého souboru, následující (*v*+1)-tý vytvořený výstupní běh uložíme opět do prvního souboru atd. Tj. vytvářené výstupní běhy pravidelně rozdělujeme do *v* výstupních souborů.

V okamžiku, kdy všechny běhy ze vstupních souborů jsou přečteny, výstupní soubory uzavřeme a otevřeme je jako vstupní a předchozí vstupní soubory uzavřeme a otevřeme je nyní jako výstupní. A znovu opakujeme proces zatřiďování. Tenhle postup probíhá tak dlouho, dokud vytvořený výstupní běh není tak dlouhý, že obsahuje všechny tříděné prvky.

Příklad. Budeme třídit prvky

3 7 5 15 12 1 11 8 17 4 19 10 2 6 9 14

Počet souborů zvolme *r*=4. Polovina je *v*=2.

Ve fázi rozdělování se vytvoří dva soubory s prvky

1. soubor: 3 5 12 11 17 19 2 9

2. soubor: 7 15 1 8 4 10 6 14

Délka běhů je na začátku 1.

Tyto soubory učiníme vstupní a necháme proběhnout zatřiďování, na jehož závěru budou vytvořeny dva soubory s běhy délky 2:

1. soubor: 3 7 | 1 12 | 4 17 | 2 6

2. soubor: 5 15 | 8 11 | 10 19 | 9 14

Opět tyto vytvořené soubory otevřeme jako vstupní a necháme na nich proběhnout zatřiďování, na jehož závěru budeme mít soubory s běhy délky 4:

1. soubor: 3 5 7 15 | 4 10 17 19

2. soubor: 1 8 11 12 | 2 6 9 14

A znovu úlohu vstupních a výstupních souborů obrátíme a znovu necháme proběhnout proces zatřiďování. Nyní vzniknou běhy délky 8:

1. soubor: 1 3 5 7 8 11 12 15

2. soubor: 2 4 6 9 10 14 17 19

A poslední průchod, při kterém vznikne běh délky 16:

1. soubor: 1 2 3 4 5 6 7 8 9 10 11 12 14 15 17 19

2. soubor: --

Zbývá popsat, jak probíhá vlastní zatřiďování *v* vstupních běhů v jeden výstupní běh.

Algoritmus zatřiďování

K zatřiďování potřebujeme tolik proměnný, kolik je vstupních souborů, tedy *v*. Tyto proměnné si označme

$$x_1, x_2, ..., x_v$$

1. Do každé z proměnných $x_1, x_2, ..., x_v$ načteme první prvek z v vstupních běhů.

2. Vybereme nejmenší z prvků v $x_1, x_2, ..., x_v$. Nechť tento prvek je třeba v proměnné x_i. Prvek zapíšeme do výstupního souboru, do kterého je právě ukládán výstupní běh, a do proměnné x_i načteme další prvek z i-tého vstupní běhu (zbývá-li v něm ještě nějaký). Tento proces pokračuje tak dlouho, dokud všechny prvky z proměnných nejsou zapsány do výstupního běhu a dokud všechny prvky ze současných v vstupních běhů nejsou vyčerpány.

Pro zatřiďování je zapotřebí jen několik proměnných.

Příklad. Budeme zatřiďovat dva vstupní běhy délky 4 z minulého příkladu

3 5 7 15

1 8 11 12

Po načtení prvních prvků do proměnných je:

x_1 = 3 v 1. běhu zbývá: 5 7 15

x_2 = 1 v 2. běhu zbývá: 8 11 12

Vybereme nejmenší prvek 1 a zapíšeme ho na výstup. Na jeho místo načteme další prvek z příslušného běhu:

x_1 = 3 v 1. běhu zbývá: 5 7 15

x_2 = 8 v 2. běhu zbývá: 11 12

Vybereme nejmenší prvek 3 a zapíšeme ho na výstup.

x_1 = 5 v 1. běhu zbývá: 7 15

x_2 = 8 v 2. běhu zbývá: 11 12

Vybereme nejmenší prvek 5 a zapíšeme ho na výstup.

x_1 = 7 v 1. běhu zbývá: 15

x_2 = 8 v 2. běhu zbývá: 11 12

Vybereme nejmenší prvek 7 a zapíšeme ho na výstup.

Atd.

Složitost metody

Vezměme nejprve, kolik operací zahrnuje operace zatřiďování pro jeden prvek:

jednu operaci čtení, kdy je prvek načten do příslušné proměnné,

v-1 operací srovnání, kdy je tento prvek vybrán jako nejmenší (případně těchto operací může být méně, když už některé vstupní běhy jsou vyčerpány a příslušné proměnné těchto běhů už žádný prvek neobsahují),

jednu operaci zápisu, kdy je prvek zapsán do souboru, do něhož je právě výstupní běh ukládán.

Uvážíme-li, že máme n tříděných prvků, pak pro jeden průchod zatřiďování dostáváme

n operaci čtení + přibližně $n*(v-1)$ operací srovnání + n operací zápisu.

Protože v je zvolená konstanta, jejíž hodnota je velmi malá vzhledem k počtu tříděných prvků n, je složitost jednoho průchodu zatřiďování lineární.

Zbývá stanovit, kolik průchodů zatřiďování proběhne. V prvním průchodu zatřiďování, kdy délka vstupních běhů je 1, vzniknou výstupní běhy délky v. V druhém průchodu zatřiďování, kdy délka vstupních běhů je v, vzniknou výstupní běhy délky $v*v$. Atd.

Průchod	Délka vstupních běhů	Délka výstupních běhů
1	1	v
2	v	v^2
3	v^2	v^3
…	…	…
k	$v^{(k-1)}$	v^k

Třídění končí, když výstupní běh obsahuje všech *n* tříděných prvků. Tedy pro pořadí *k* posledního průchodu dostáváme vztah

$$v^k = n \quad .$$

A odtud použitím logaritmu

$$k = \log_v(n) \quad .$$

Složitost vnějšího třídění je optimální.

Počet průchodů logaritmicky závisí na počtu tříděných prvků. Když to vynásobíme lineární složitostí jednoho průchodu, dostáváme, že složitost popsané metody vnějšího třídění je *Θ(n*log(n))*. Už jsme se zmínili, že tato složitost je pro třídění nejmenší možná, tedy optimální.

3.4.2 Vnější třídění s využitím vnitřního třídění

Vnější třídění lze zefektivnit využitím vnitřního třídění.

V předchozím popisu vnějšího třídění se ve fázi rozdělování vytvářely běhy obsahující jen jeden prvek. Zde se nabízí myšlenka využít některé metody vnitřního třídění a ve fázi rozdělování vytvářet běhy delší než 1. Pak by zřejmě bylo zapotřebí méně průchodů zatřiďování a celkový čas pro třídění by se tím zkrátil. Tento postup budeme aplikovat tak, že stanovíme nějaký počet prvků *m*, který se ještě vejde do paměti. Při rozdělování vždy do paměti načteme *m* prvků, ty v ní setřídíme, přirozeně některou z efektivních metod (Quicksort, třídění haldou), a po setřídění je zapíšeme do příslušného výstupního souboru jako jeden běh. Nyní pro počty běhů při následných průchodech zatřiďování bude platit

Průchod	Délka vstupních běhů	Délka výstupních běhů
1	m	$m * v$
2	$m * v$	$m * v^2$
3	$m * v^2$	$m * v^3$
…	…	…
k	$m * v^{(k-1)}$	$m * v^k$

Odtud pro poslední *k*-tý běh dostáváme vztah

$$m * v^k = n \quad .$$

A použitím logaritmu

$$k = \log_v(n) - \log_v(m) \quad .$$

Z toho je zřejmé, že čím bude délka běhů *m* vytvářených vnitřním tříděním ve fázi rozdělování větší, tím více ušetříme průchodů zatřiďování. To se dalo i očekávat.

3.4.3 Polyfázové třídění

Vraťme se ke vztahu, jež udává počet průchodů zatřiďování:

$$k = \log_v(n) - \log_v(m) \ .$$

Polyfázové třídění promyšleným způsobem využívá celkový počet souborů.

Je zřejmé, že počet potřebných průchodů zatřiďování mimo jiné závisí na základu logaritmu *v*, tj. počtu vstupních souborů. Čím větší tento bude, tím méně průchodů bude potřeba. Můžeme sice počet vstupních souborů zvýšit zvolením větší počtu celkově používaných souborů, ale na druhé straně příliš mnoho používaných souborů má negativní vliv na výkonnost systému. Spíše se nabízí myšlenka celkový počet *r* používaných souborů využít efektivněji a to tak, že místo rozdělení

$$\frac{r}{2} \text{ vstupních souborů } + \frac{r}{2} \text{ výstupních souborů}$$

použít rozdělení

$$r-1 \text{ vstupních souborů } + \text{ 1 výstupní soubor .}$$

Při tomto rozdělení musí mít vstupní soubory rozdílné počty běhů, aby jeden z nich se vyčerpal dříve než ostatní. V tom okamžiku se vyčerpaný vstupní soubor uzavře a stane se nyní novým výstupním souborem a dosavadní výstupní soubor se rovněž uzavře a přidá se ke vstupním souborům. Aby tento princip fungoval po celou dobu třídění, musí ve fázi rozdělování být vstupní běhy rozděleny do *r*-1 vytvářených souborů ve specifických počtech.

Vezměme například celkový počet souborů *r*=4. Při odvození, kolik běhů musí být na začátku ve fázi rozdělování uloženo do jednotlivých souborů, vyjdeme ze situace, která je na konci třídění – 3 současně vyčerpané vstupní soubory a 1 běh na výstupním souboru. Což znamená, že než začalo ukládání běhů na výstupní soubor, musel na všech 3 vstupních soubor v této chvíli zbývat právě jeden běh. Když budeme touto úvahou pokračovat dál dostaneme:

Soubor číslo 1	2	3	4	Celkem běhů	Průchod zatřiďování
0	0	0	1	1	Výsledný stav
1	1	1	0	3	k (poslední)
2	2	0	1	5	k-1 (předposlední)
4	0	2	3	9	k-2
0	4	6	7	17	k-3
7	11	13	0	31	k-4
20	24	0	13	57	k-5
44	0	24	37	105	k-6

Vezměme například stav na začátku průchodu *k*-2. V tomto okamžiku v 1. souboru zbývají 4 nepřečtené běhy, 2. soubor je v této chvíli zcela přečten, stane se tudíž v tomto průchodu výstupní, v 3. souboru zbývají 2 běhy k přečtení a ve 4. souboru jsou ještě 3 nepřečtené běhy. Což znamená, že nejvíce nepřečtených běhů je v 1. souboru, ten tedy byl vytvořen v předchozím třídícím průchodu *k*-3. Každý z běhů ve vytvořeném 1. souboru byl vytvořen zatříděním běhů ze zbývajících tří souborů. Pro 4 vytvořené běhy byly tedy z každého ze vstupních souborů přečteny 4 běhy. Tedy každý z těchto souborů na začátku průchodu *k*-2 měl o 4 nepřečtené běhy více než na jeho konci. 2. soubor na konci tohoto průchodu už nemá žádný nepřečtený běh, tedy předtím měl 4 nepřečtené, 3. soubor má na konci 2 nepřečtené, tudíž předtím musel mít 6 nepřečtených běhů atd.

Ve fázi rozdělování je nutné tyto počty dodržet. Jestliže například vnitřním tříděním získáme 50 běhů, pak zvolíme k tomu nejbližší možný celkový počet běhů, což je 57. Do prvního souboru dáme 24 běhů, do druhého 20 běhů, do třetího zbývajících 6 běhů, které do požadovaného počtu 13 běhů doplníme 7 fiktivními běhy, což jsou běhy nulové délky (neobsahující žádný prvek).

Příklad. Budeme opět třídit prvky

3 7 5 15 12 1 11 8 17 4 19 10 2 6 9 14

Bez vnitřního třídění máme 16 vstupních běhů po jednom prvku. Tedy zvolíme celkový počáteční počet běhů 17 a rozdělíme je do tří souborů následovně:

1. soubor: 3 | 7 | 5 | 15 | 12 | 1 | 11

2. soubor: 8 | 17 | 4 | 19 | 10 | 2

3. soubor: 6 | 9 | 14 | fiktivní

4. soubor: --

Po prvním průchodu zatřiďování

1. soubor: 12 | 1 | 11

2. soubor: 10 | 2

3. soubor: --

4. soubor: 3 6 8 | 7 9 17 | 4 5 14 | 15 19

Po druhém průchodu zatřiďování

1. soubor: 11

2. soubor: --

3. soubor: 3 6 8 10 12 | 1 2 7 9 17

4. soubor: 4 5 14 | 15 19

Po třetím průchodu zatřiďování

1. soubor: --

2. soubor: 3 4 5 6 8 10 11 12 14

3. soubor: 1 2 7 9 17

4. soubor: 15 19

Ve všech třech souborech, které budou v následujícím průchodu zatřiďování vstupní, je už jen jeden běh, čímž tento průchod bude poslední a na jeho konci bude v prvním souboru setříděná posloupnost.

Popsaná metoda rozdělování běhů se nazývá polyfázové třídění. Spolu s použitím vnitřního třídění pro vytváření počátečních běhů je běžně používána v třídících programech pro vnější třídění.

Kontrolní otázky

1. *Jakým způsobem vytváříme z více vstupních běhů jeden výstupní běh?*
2. *Co je principem polyfázového třídění?*
3. *Jaká je časová složitost vnějšího třídění?*

Cvičení

1. Ukažte, jak proběhne vnějším tříděním bez použití vnitřního třídění a se dvěma vstupními a dvěma výstupními soubory pro následující písmena.

C	J	F	I	H	E	B	A	L	D	H	L	B	K	F

2. Jak bude probíhat zatřídění tří následujících běhů délky 3:

B	G	I

A	C	F

D	E	H

3. Máme setřídit 100000 záznamů. V paměti dokážeme třídit 2000 záznamů. Použijeme celkově 5 souborů. Budeme třídit polyfázovým tříděním. Napište, kolik bude na začátku na 4-ti vstupních souborech běhů.

Úkoly k textu

Při polyfázovém třídění nemají běhy v jednotlivých vstupních souborech stejnou délku. Například používáme-li 3 vstupní soubory a počáteční délka běhů je *m*, pak na začátku je délka běhů ve všech souborech stejná

m m m .

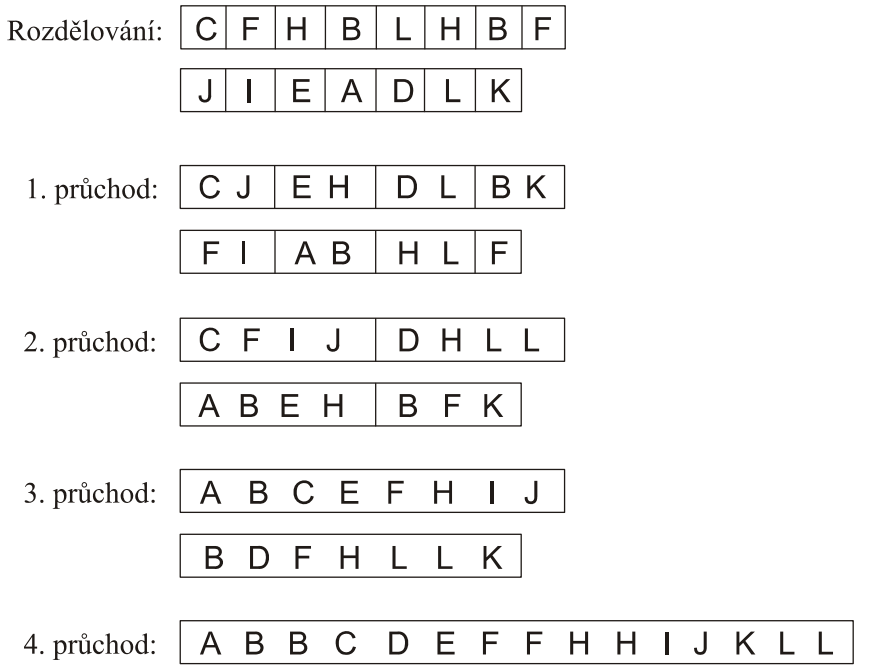
Při prvním průchodu z nich vznikají běhy délky 3*m*. Po vyčerpání prvního souboru a následném zařazení výstupního souboru na vstup už délky budou

3m m m .

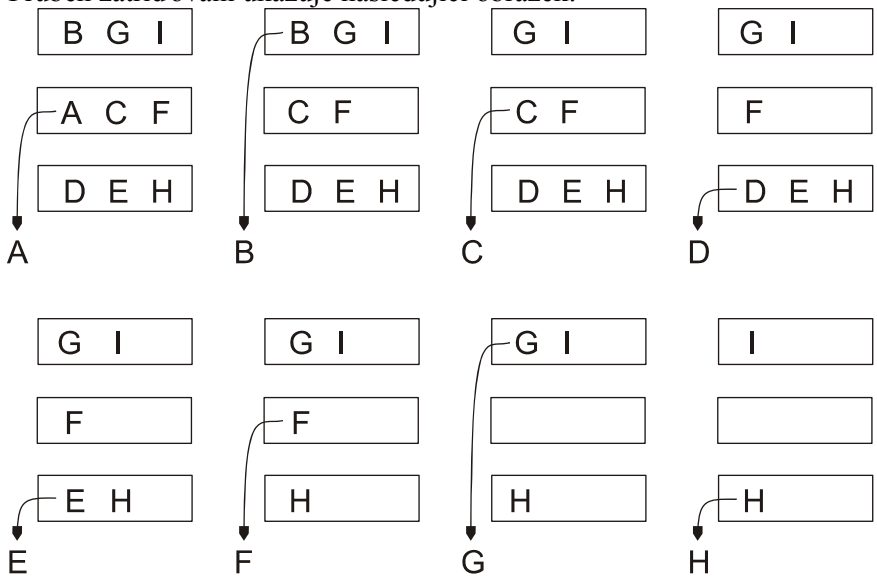
Zamyslete se nad tím, jak to postupně bude vypadat po vyčerpání dalších vstupních souborů.

Řešení

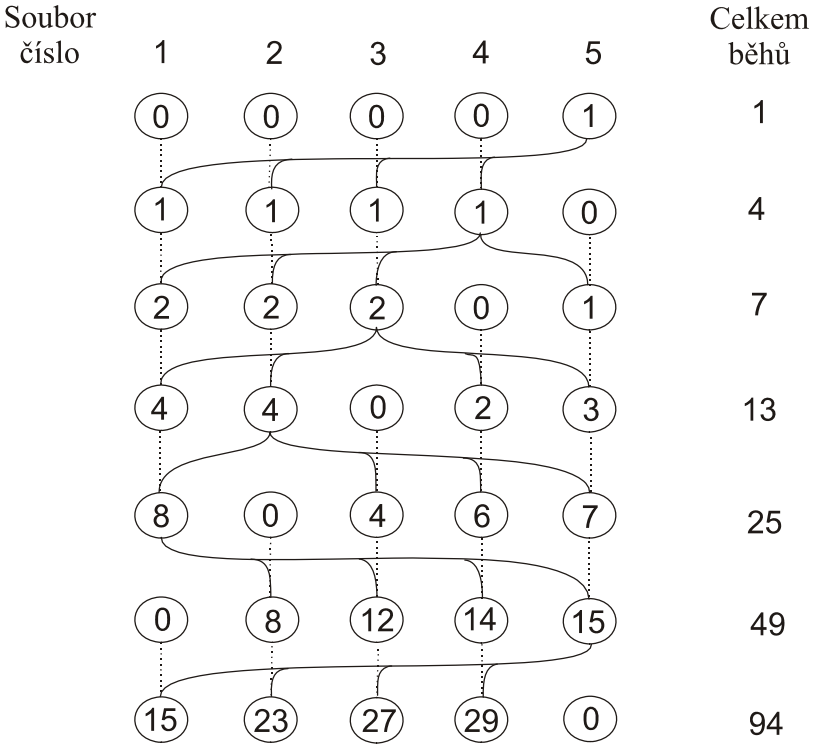
1. Postup rozdělování a následné zatřiďování ukazuje následující obrázek:



2. Průběh zatřiďování ukazuje následující obrázek:



3. Při vnitřním třídění po 2000 prvcích dostaneme pro 100000 vstupních prvků 50 běhů. Odvodíme nejmenší počet vstupních běhů pro polyfázové třídění, který je roven 50 nebo větší než 50.



Z předchozího obrázku plyne, že musíme zvolit 94 počátečních běhů – 50 získaných tříděním + 44 fiktivních.

4 Vyhledávání

Vyhledávání je další velmi důležitou a často se vyskytující úlohou. Při ní máme opět zadanou nějakou množinu prvků a cílem je nalézt mezi nimi takový prvek, který má danou hodnotu, anebo případně zjistit, že takový prvek mezi nimi není.
Máme tedy *n* prvků dat

Potřeba najít nějaký údaj nebo hodnotu je v praxi dost častá.

$$a_1, a_2, \ldots a_n \quad ,$$

dále máme dánu hodnotu prvku, označme ji x (nebo hodnotu klíče prvku, hledáme-li mezi strukturovanými prvky, tj. záznamy) a hledáme prvek a_i, pro který platí

$$x = a_i \quad .$$

4.1 Vyhledávání v lineární datové struktuře

Studijní cíle: Tato část vysvětluje principy vyhledávání v lineárních datových strukturách a poskytuje jejich podrobný popis včetně základního nástinu, jak je implementovat.

Klíčová slova: Vyhledávání, zarážka, binární vyhledávání.

Potřebný čas: 1 hodina 30 minut

Mezi nejjednodušší případy vyhledávání patří vyhledávání v lineární datové struktuře, tj. v poli nebo v seznamu. Přepokládáme přitom, že prvky jsou v ní uloženy v libovolném pořadí (nesetříděné). Není zde jiný způsob, než prvky postupně procházet (třeba od začátku směrem ke konci) a každý srovnat s hledanou hodnotou. Počet srovnání se přitom pohybuje od 1, jestliže hledaný prvek je hned první, po *n*, jestliže hledaný prvek je až poslední anebo hledaný prvek mezi prohledávanými prvky není obsažen (nebyl nalezen). Tedy

Hledání v poli je sice velmi jednoduché, nicméně málo efektivní.

Průměrný počet srovnání (je-li prvek nalezen) $= \frac{n+1}{2}$

Maximální počet srovnání $= n$

Jde-li o vyhledávání v poli, je kód programu v tomto případě tak jednoduchý, že si ho můžeme uvést (v jazyce C++, C#). Hledaná hodnota nechť je v proměnné *x* a jako index použijeme proměnnou *i*. Předpokládáme, že index prvního prvku pole je 0.

```
i = 0;
for (;i<n;i++) if (x==a[i]]) break;
if (i<n) { /* prvek byl nalezen*/ }
```

Je zřejmé, že v cyklu se kromě srovnání, zda prvek je roven hledané hodnotě, rovněž vždy dělá srovnání, zda už nejsme na konci pole (podmínka `i<n`). To lze odstranit a vyhledávání urychlit tak, že pole s prvky vytvoříme o jeden prvek větší a jako poslední prvek v poli před každým prohledáváním dáme hledanou hodnotu (užívá se pro ni označení zarážka). Pak cyklus hledání vždy nalezne hledaný prvek a bude jen zapotřebí zjistit, zda tento prvek byl nalezen ještě před zarážkou.

```
a[n] = x;
i = 0;
for (;;i++) if (x==a[i]]) break;
if (i<n) { /* prvek byl nalezen */ }
```

4.2 Binární vyhledávání v poli

V případě pole je mnohem příznivější případ pro vyhledávání, když prvky jsou v něm uspořádány (seřazeny) dle velikosti. Tj. platí pro ně

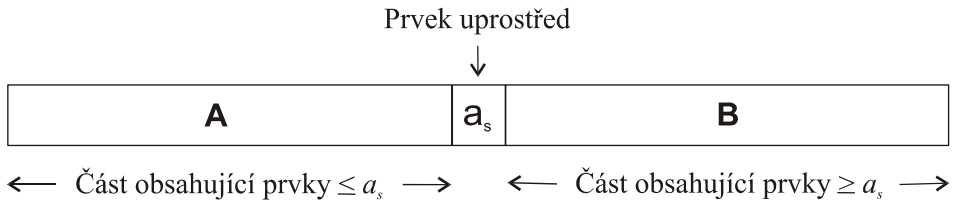
Pro vyhledávání v poli je výhodné, když je setříděné.

$$a_1 \le a_2 \le ... \le a_{n-1} \le a_n$$

Zde se dá použít algoritmus binárního vyhledávání, často také nazývaný vyhledávání půlením intervalu.

Popis algoritmu

Vezmeme prvek, který je v poli uprostřed (je-li počet prvků sudý, jsou uprostřed dva prvky, zde vezmeme jeden z nich - při implementaci metody to zpravidla vychází tak, že se bere ten první), označme jeho index *s*.



Následně provedeme srovnání hledané *x* hodnoty s hodnotou středního prvku a_s:

Nejprve srovnáme, zda je $x < a_s$:

Pokud ano, pak zřejmě hledaný prvek, pokud v poli vůbec je, musí být v části *A*, jež je nalevo od středního prvku a_s. Je-li část *A* neprázdná (obsahuje aspoň jeden prvek), rekurzivně na ni provedeme stejný postup. Je-li už prázdná, vyhledávání neúspěšně končí. Hledaný prvek není v poli obsažen.

Pokud neplatí $x < a_s$, provedeme další srovnání. Srovnáme, zda je $x > a_s$:

Pokud ano, musíme v dalším kroku hledání pokračovat v části *B*, jež je napravo od středního prvku a_s. Je-li část *B* neprázdná (obsahuje aspoň jeden prvek), rekurzivně na ni provedeme stejný postup. Je-li už prázdná vyhledávání neúspěšně končí.

Pokud není $x > a_s$, zbývá už jen možnost, že platí $x = a_s$, čímž vyhledávání končí, neboť prvek a_s je hledaným prvkem.

Příklad. Nechť v poli jsou prvky

1 3 5 8 12 15 21 24 32 40

A máme najít prvek *x*=15.

Střední prvek je číslo 12:

1 3 5 8 | 12 | 15 21 24 32 40

A protože hledaná hodnota je větší, pro další krok vezmeme část napravo

15 21 24 32 40

Střední prvek je číslo 24:

15 21 | 24 | 32 40

A protože hledaná hodnota je menší, vezmeme část nalevo

15 21

Střední prvek je nyní číslo 15:

A střední prvek je zde roven hodnotě v proměnné *x*, čímž je i hledaným prvkem.

Složitost metody

Při odvození složitosti vyjdeme z délky prohledávané části pole. V prvním kroku začínáme celým polem, tedy *n* prvky. Pokud v něm hledaný prvek nebyl nalezen, pak do dalšího kroku vezmeme část nalevo od něho nebo napravo od něho. Délka částí je $\frac{n-1}{2}$ nebo $\frac{n}{2}$ podle toho, zda počet prvků *n* je lichý nebo sudý. Pro odvození vezmeme ten „horší“ případ, že délka části vybrané pro následující krok je $\frac{n}{2}$.

Krok	Délka prohledávané části
1.	n
2.	$\frac{n}{2}$
3.	$\frac{n}{2^2}$
4.	$\frac{n}{2^3}$
….	
k-tý	$\frac{n}{2^{k-1}}$

Zřejmě vyhledávání skončí nejpozději v kroku, kdy délka prohledávané části je už tvořena jen jedním prvkem. Tedy položme

$$1 = \frac{n}{2^{k-1}} \quad .$$

Úpravou

$$2^{k-1} = n \quad .$$

A použitím funkce logaritmu

$$k = \log_2(n) + 1 \quad .$$

Binární vyhledávání má vynikající složitost.

Maximální počet kroků logaritmicky závisí na počtu prvků v prohledávané posloupnosti. V každém kroku provádíme nejvýše dvě operace srovnání. První operací zjistíme, zda hledaný prvek je menší než střední prvek. Pokud ano, pokračujeme v hledání v části nalevo. Pokud ne, druhou operací srovnání zjistíme, zda hledaný je větší než střední prvek, čímž rozhodneme, zda pokračovat v hledání v části napravo anebo už jsme hledaný prvek našli.

Závěr: Složitost binárního vyhledávání je $\Theta(\log(n))$. Z ní plyne, že binární vyhledávání je mnohem rychlejší než vyhledávání, kdy prvky nejsou uspořádány.

Průvodce studiem

Možná jste si nyní vzpomněli na třídění přímým vkládáním. Zde se místo pro vložení hledalo v setříděné části posloupnosti. Dalo by se tedy použít mnohem rychlejší binární vyhledávání. Na časovou složitost metody by to ale nemělo vliv, neboť tu zůstává ještě operace posunutí a ta má kvadratickou složitost.

Kontrolní otázky

1. *Proč a jakým způsobem používáme zarážku při vyhledávání v nesetříděném poli?*
2. *Co je principem binárního vyhledávání?*
3. *Jaký je rozdíl časové složitosti při vyhledávání v nesetříděném a setříděném poli?*

Cvičení

1. Ukažte po jednotlivých krocích, jak proběhne binární vyhledání písmena K následujícím poli písmen.

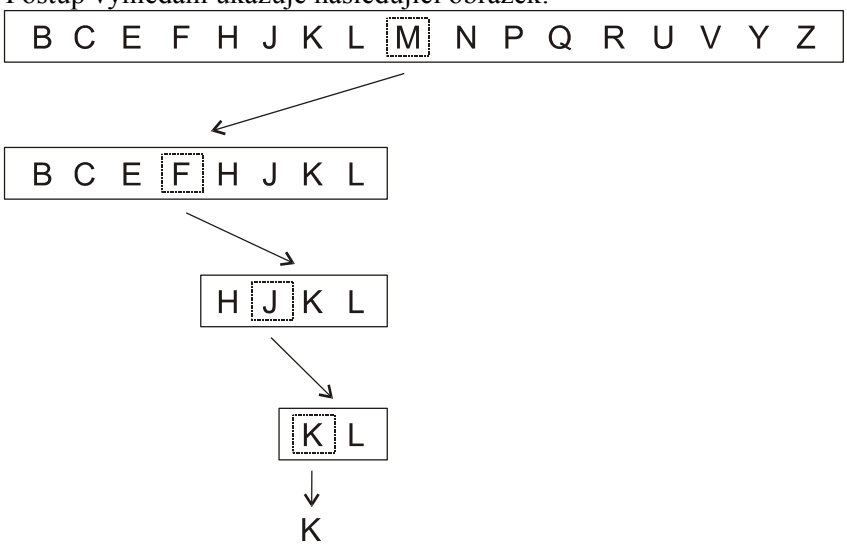
B	C	E	F	H	J	K	L	M	N	P	Q	R	U	V	Y	Z

Úkoly k textu

Třídění je časově náročnější než vyhledávání. Proto v případě pole nesetříděných údajů se nevyplatí ho napřed setřídit kvůli tomu, abychom pak mohli použít efektivnější binární vyhledávání. Nicméně v případě, že bychom v něm hledali velmi často, mohlo by to už být účelné ho napřed setřídit. Máme-li *n* prvků, kolikrát v nich musíme hledat, aby bylo rozumné je napřed setřídit?

Řešení

1. Postup vyhledání ukazuje následující obrázek:



4.3 Binární vyhledávací stromy

Studijní cíle: Tato část vysvětluje princip binárních vyhledávacích stromů, popisuje konstrukci AVL stromů, ukazuje, jak do takového stromu přidat prvek nebo z něho prvek odebrat a zejména, jak pak provést transformace, jimiž se zajistí zachování vyváženosti stromu.

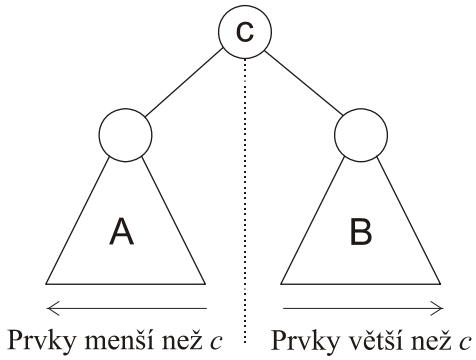
Klíčová slova: Vložení, odebrání, AVL strom.

Potřebný čas: 2 hodiny

Pro případy, kdy hodnoty jsou průběžně přidávány nebo rušeny, máme vyhledávací stromy.

Binární vyhledávání popsané v předchozí části má velmi příznivou časovou složitost. Pokud bychom měli množinu prvků, v níž velmi často a intenzívně hledáme, pak by se zřejmě vyplatilo je na začátku setřídit. Problém ovšem nastane, když tato množina se v průběhu času mění, tj. jsou k ní přidávány nové prvky nebo z ní naopak některé prvky jsou odebírány. Už jsme uvedli, že vkládání prvků doprostřed pole nebo jejich odebírání zprostředka pole je poměrně neefektivní operace, neboť je spojena s přesuny poměrné značné části prvků v poli. Pro takovéto případy je výhodnější použít vyhledávací stromy. Nejjednodušší z nich jsou binární vyhledávací stromy.

Binární vyhledávací strom má každém uzlu jeden prvek (na obrázku je tento prvek označen *c*). Dále platí, že prvek v jeho levém následníku a rovněž i všechny prvky v celém levém podstromu (na obrázku podstrom *A*), který tímto následníkem začíná, jsou menší než prvek *c*. A naopak všechny prvky v pravém podstromu (na obrázku *B*) jsou větší než prvek *c*.



Postup vyhledávání

1. Počáteční krok

Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat *u*. Na začátku jím bude kořen stromu.

Hledaná hodnota nechť je *x*.

2. Průběžný krok

Vezmeme prvek obsažený v aktuálním uzlu *u*, označme ho *c*, a provedeme jeho srovnání s hledanou hodnotou *x*:

Nejprve srovnáme, zda je $x<c$:

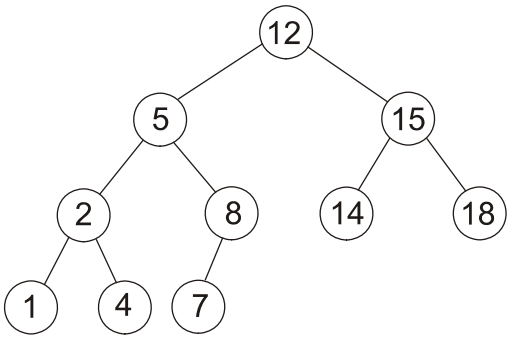
Pokud ano, pak je nutné v hledání pokračovat v levém podstromu. Jako nový aktuální uzel *u* položíme levého následníka současného aktuálního uzlu a znovu provedem krok 2. Pokud současný aktuální uzel levého následníka nemá, vyhledávání končí - hledaný prvek není ve stromu obsažen.

Pokud není $x<c$, srovnáme, zda je $x>c$:

Pokud ano, je nutné v hledání pokračovat v pravém podstromu. Jako nový aktuální uzel *u* položíme pravého následníka současného aktuálního uzlu a opět provedeme krok 2. Pokud uzel pravého následníka nemá, vyhledávání končí, hledaný prvek není ve stromu obsažen.

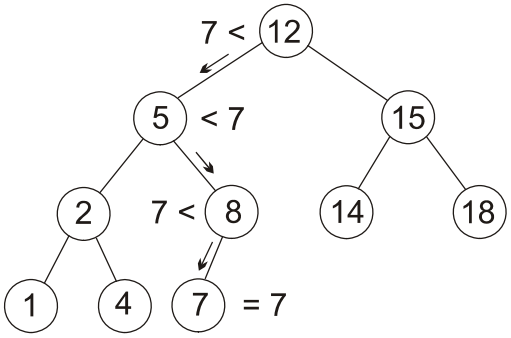
Pokud není $x>c$, zbývá už jen případ, že platí $x=c$, čímž jsme u konce, neboť prvek obsažený v současném aktuálním uzlu *u* je tím hledaným prvkem.

Příklad. Mějme binární vyhledávací strom



A hledáme v něm hodnotu x=7.

Na začátku aktuální uzel je kořen u=12. Srovnáme, zda je $x<u$. To platí (7<12) a nový aktuální uzel bude levý následník u=5. A opět srovnáme, zda $x<u$. To nyní neplatí a proto srovnáme, zda je $x>u$. To platí (7>5) a nový aktuální uzel bude pravý následník u=8. A znovu srovnáme, zda je $x<u$. To platí (7<8) a nový aktuální uzel bude levý následník u=7. A opět srovnáme, zda $x<u$. To neplatí a proto srovnáme, zda je $x>u$. Ani to neplatí, čímž je hledaný prvek nalezen, je v současném aktuálním uzlu. Celý postup ukazuje následující obrázek.



Efektivnost vyhledávacího stromu závisí na jeho výšce.

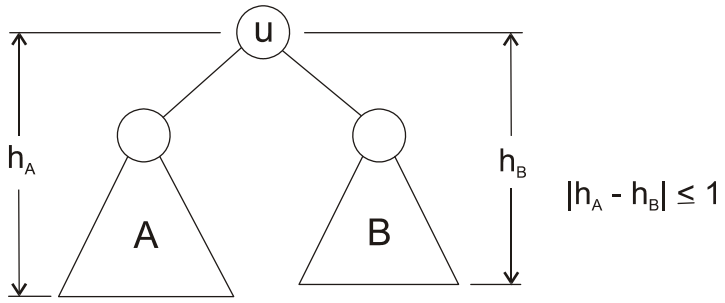
Počet srovnání potřebný k nalezení prvku je závislý na délce cesty od kořene k danému prvku. Maximální délka cesty je dána výškou stromu. V tomto ohledu je optimálním řešením vyvážený binární vyhledávací strom, který svou logaritmickou závislostí výšky stromu na počtu uzlů ve stromu zajišťuje i logaritmickou složitost vyhledávání $\Theta(log(n))$. Nicméně ve vyváženém binárním vyhledávacím stromu je poměrně složité provádět operace přidání nebo odebrání prvku, neboť přidání prvku reprezentuje přidání uzlu na určité místo ve stromu a odebrání prvku reprezentuje odebrání uzlu z určitého místa ve stromu. U obou těchto operacích se musí při nich provést přestavba určité části stromu tak, aby strom po nich byl opět vyvážený. A to je časově značně náročné. Proto se v praxi používají AVL stromy, které mají sice nižší stupeň vyvážení a tudíž vyžadují o něco delší čas pro vyhledávání, ale na druhé straně operace přidání a odebraní prvku a následné obnovení vyvážení probíhají u nich mnohem rychleji.

4.3.1 AVL stromy

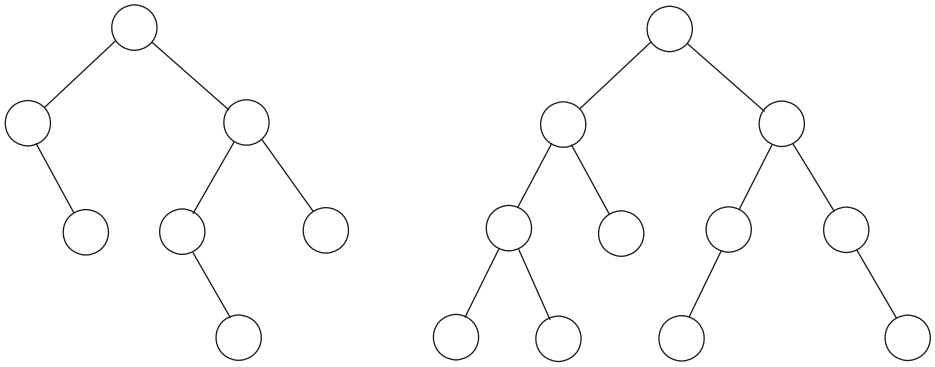
Úplně vyvážit binární vyhledávací strom je značně komplikované. Řešením jsou AVL stromy.

AVL stromy jsou binární vyhledávací stromy, u nichž je zajištěna taková míra vyváženosti, že na jedné straně poskytuje operaci vyhledávání dobrou složitost a na druhé straně umožňuje poměrně rychle obnovit vyvážení stromu po přidání nebo odebrání prvku ze stromu. Je to tedy určitý kompromis mezi vyvážeností a obtížností operací přidání a odebrání prvku.

Vyváženost v AVL stromu je zajištěna podmínkou, že pro každý uzel u stromu musí platit, že rozdíl mezi výškou jeho levého podstromu a výškou jeho pravého podstromu je nejvýše 1. Přitom výškou podstromu zde rozumíme maximum ze vzdáleností od uzlu u k uzlům podstromu, tedy vzdálenost mezi uzlem u a uzly, které jsou úplně naspodu podstromu. Pokud uzel nemá daného následníka (levého, pravého), je výška tohoto podstromu 0.



Na následujícím obrázku jsou dva AVL stromy výšky 3. Přitom v levém stromu mají všechny uzly (vyjma listů) podstromy rozdílné délky, čímž takový AVL strom má při dané výšce nejmenší možný počet uzlů. Tedy AVL strom výšky 3 má nejméně 7 uzlů (nejvíce jich má 15).



Průvodce studiem

Název AVL nemá žádný mnemonický význam. Je odvozen od jmen svých tvůrců, kterými jsou dva rusové Adelson-Velskii a Landis.

Postup vyhledávání

AVL strom je z pohledu vyhledávání jen specifickým případem obecného binárního vyhledávacího stromu a tudíž algoritmus vyhledávání je stejný.

Postup přidání prvku

Do AVL stromu lze snadno přidat nový prvek.

Operace přidání prvku do AVL znamená na příslušném místě přidat do AVL stromu uzel, do kterého nový prvek vložíme, a následně ověřit, zda nedošlo k narušení vyvážení stromu, a pokud ano, strom znovu vyvážíme.

1. Přidání prvku

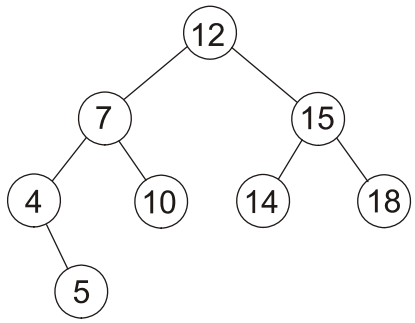
Označme přidávaný prvek *x*. Provedeme jeho vyhledání ve stromu. Použijeme k tomu již popsaný algoritmus vyhledávání. Ten může skončit třemi způsoby:

a) Prvek *x* byl ve stromu nalezen. Tím přidávání končí, neboť prvek *x* už je ve stromu obsažen a u vyhledávacích stromů se nepředpokládá vícenásobný výskyt stejného prvku.

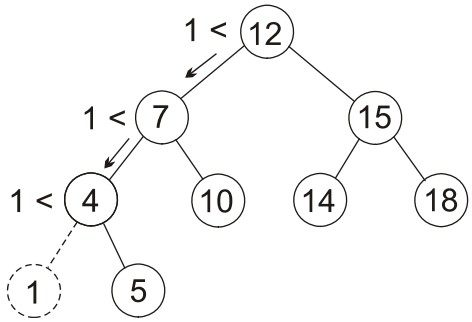
b) Vyhledávání skončilo v uzlu *u* s prvkem *c*, přičemž $x<c$ a přitom uzel *u* už nemá levého následníka. V tom případě přidáme ke stromu nový uzel jako levého následníka uzlu *u* a do něho nový prvek *x* vložíme.

c) Vyhledávání skončilo v uzlu *u* s prvkem *c*, přičemž $x>c$ a přitom uzel *u* už nemá pravého následníka. V tom případě přidáme ke stromu pravého následníka uzlu *u*, do kterého nový prvek *x* vložíme.

Příklad. Do následujícího AVL stromu



máme vložit prvek 1. Následující obrázek ukazuje postup.



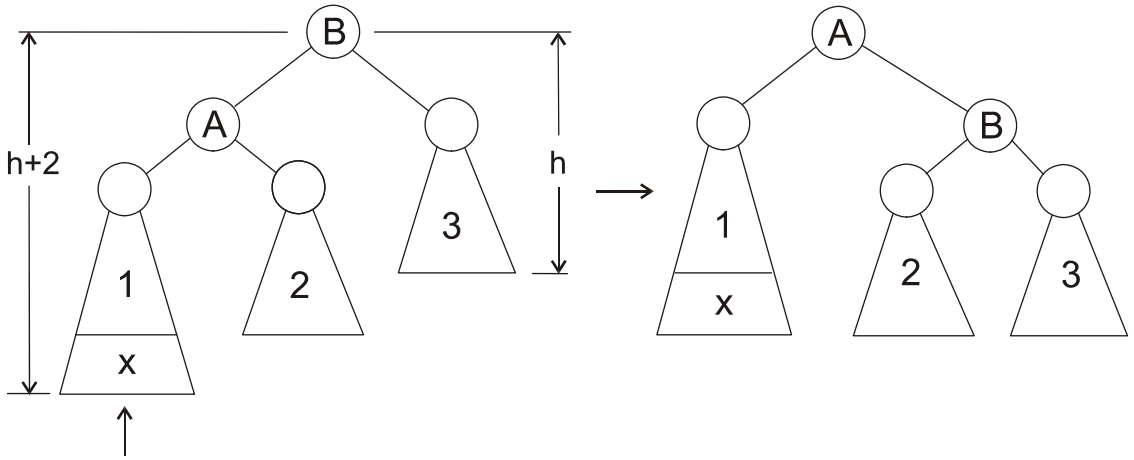
Prvek 1 bude přidán jako nový levý následník uzlu s prvkem 4.

2. Obnovení vyvážení uzlu

Vyvážení AVL stromu obnovíme lokálními transformace tvaru stromu.

Po přidání může dojít k porušení vyvážení AVL stromu. V těchto případech je nutné vhodnou transformací strom znovu vyvážit. Jako první je nutné najít uzel, u něhož došlo porušení podmínky, že výška jeho podstromů se může lišit nejvýše o 1. Proces hledání začneme u uzlu, do kterého jsme přidali následníka, a pokračujeme směrem nahoru až buďto najdeme nevyvážený uzel anebo skončíme v kořenu a tím zjistíme, že strom i po přidání zůstal vyvážený.

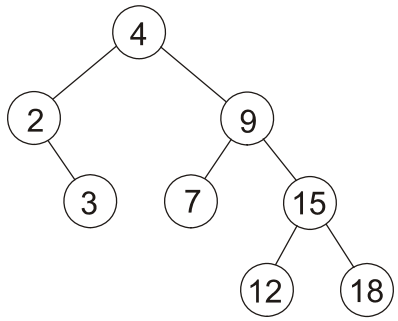
U nevyváženého uzlu mohou nastat dva základní případy. První z nich ukazuje následující obrázek. Nevyvážený uzel je v něm označen *B*.



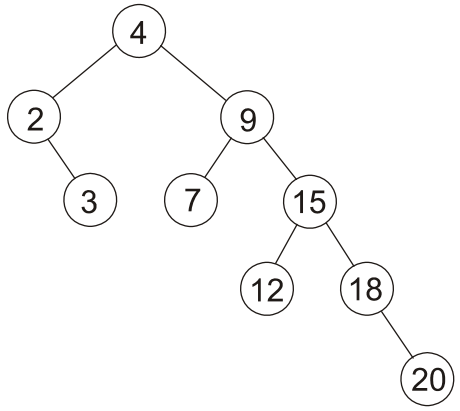
Prodloužení podstromu 1 po přidání prvku *x*

Transformace uvedená na předchozím obrázku se používá rovněž i pro zrcadlový případ (otočený kolem svislé osy).

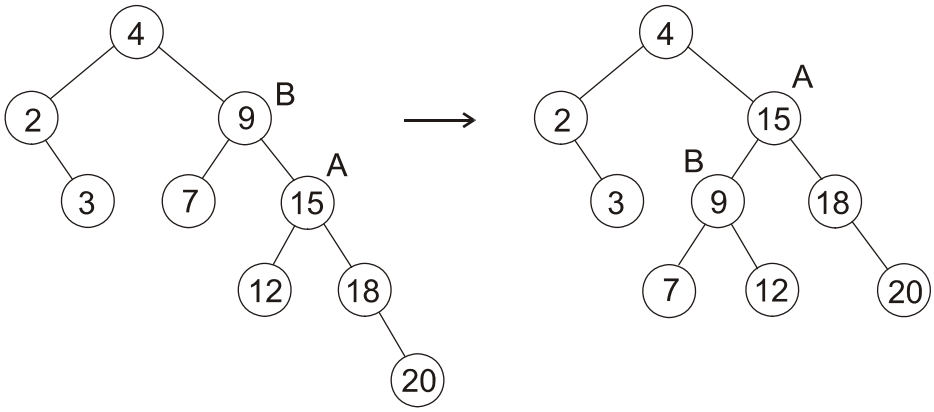
Příklad. Do následujícího AVL stromu



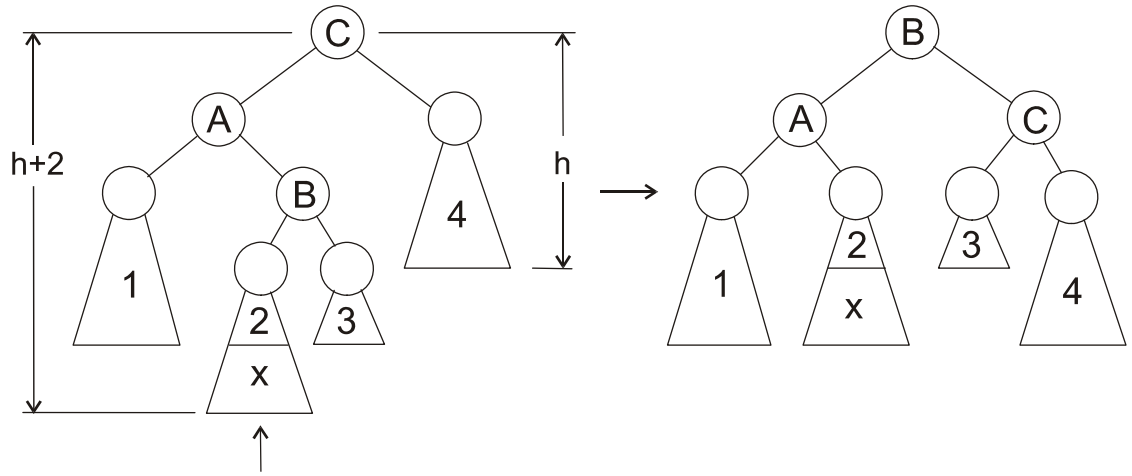
přidáme prvek 20.



Po přidání začneme směrem nahoru procházet uzly počínaje uzlem 18, do kterého byl přidán následník s novým prvkem. Uzel 18 je vyvážený, uzel 15 je vyvážený, uzel 9 ale není vyvážený, neboť jeho levý podstrom má výšku 1, zatímco pravý podstrom má výšku 3. Viditelně jde o zrcadlový případ již popsaného typu nevyvážení a k jeho odstranění použijeme transformaci u tohoto nevyvážení uvedenou.



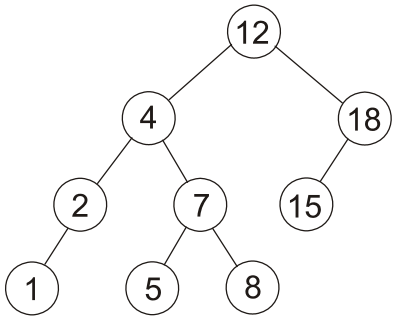
Druhý případ nevyvážení ukazuje následující obrázek. Nevyvážený uzel je v něm označen *C*.



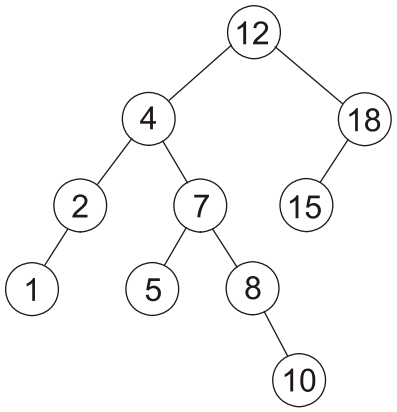
Prodloužení podstromu 2 po přidání prvku *x*

Tento typ transformace platí ještě pro variantu uvedeného případu, kdy místo podstromu 2 dojde k prodloužení podstromu 3 a rovněž pro zrcadlové případy.

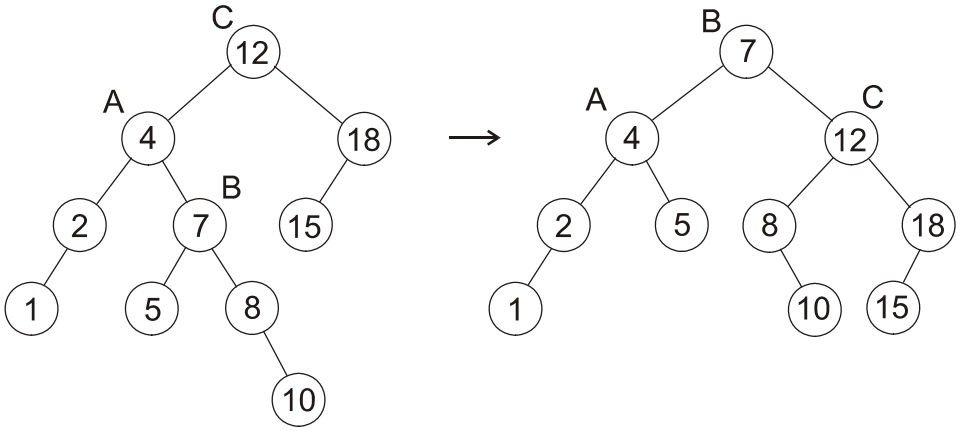
Příklad. Do následujícího AVL stromu



přidáme prvek 10.



Budeme-li nyní procházet uzly od uzlu s prvkem 8 směrem nahoru, zjistíme, že nevyvážený uzel je až kořen. Jde o případ nevyvážení, který jsme právě uvedli, a použijeme transformaci u tohoto případu uvedenou.



Postup odebrání prvku

V AVL stromu lze poměrně snadno i odstranit prvek.

Operace odebrání prvku z AVL stromu zahrnuje jeho odstranění, případné přesuny k zaplnění nelistových volných uzlů až po dosažení volného listu, který zrušíme a následně ověříme, zda nedošlo k narušení vyvážení stromu, a pokud ano, strom znovu vyvážíme.

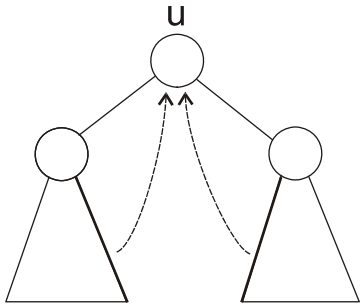
1. Odebrání prvku

Označme odebíraný prvek *x*. Provedeme jeho vyhledání ve stromu. Použijeme k tomu běžný algoritmus pro vyhledání. Ten může skončit třemi způsoby:

a) Prvek *x* nebyl ve stromu nalezen - není co odebrat, protože prvek *x* ve stromu není.

b) Prvek *x* byl nalezen v listovém uzlu. V tom případě list s tímto prvkem zrušíme.

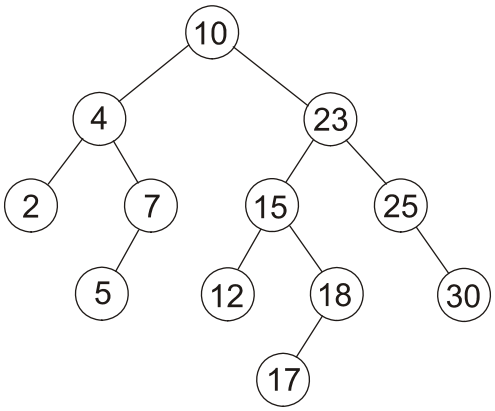
c) Prvek *x* byl nalezen v uzlu *u*, který není listem. Prvek *x* z uzlu *u* odstraníme a na volné místo v uzlu přesuneme buďto největší (a zároveň nejpravější) prvek z jeho levého podstromu anebo nejmenší (nejlevější) prvek z jeho pravého podstromu. Pokud uzel, z kterého jsme prvek přesunuli, je list, tak tento list zrušíme, jinak na tento uzel aplikujeme stejný postup. Opakováním tohoto postupu na nelistové uzly se dříve nebo později dostaneme k listu, který následně zrušíme.



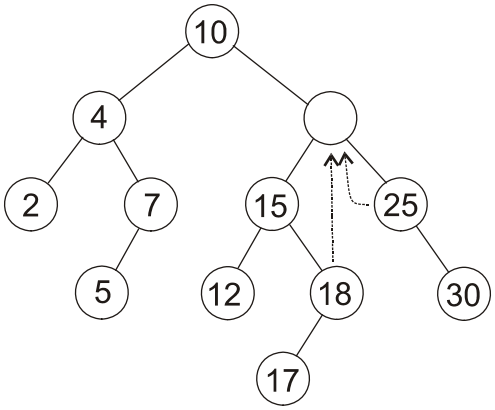
2. Obnovení vyvážení uzlu

Po odstranění listu může dojít k porušení vyvážení AVL stromu. Opět projdeme uzly směrem nahoru počínaje předchůdcem odstraněného listu a budeme ověřovat, zda u některého z nich není porušena podmínka vyváženosti AVL stromu. Pokud ano, obnovíme vyvážení některou z již uvedených transformací.

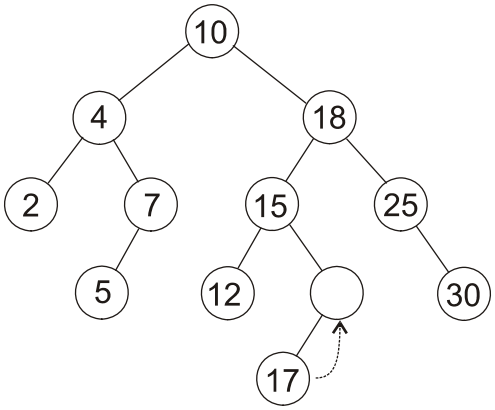
Příklad. V následujícím AVL stromu



máme zrušit prvek 23. Po jeho odebrání jsou následující dvě možnosti, jak prázdný uzel nyní zaplnit. Buďto prvkem 18, který je největším prvkem v levém podstromu, anebo prvkem 25, který je nejmenším prvek v pravém podstromu.



Zvolíme třeba zaplnění prvkem 18. Po přesunutí prvku 18 máme nyní prázdný jeho původní uzel. U tohoto uzlu je jen jedna možnost, jak ho nyní zaplnit, a to prvkem 17 z jeho levého podstromu.



Po přesunutí prvku 17 máme nyní prázdný jeho původní uzel. Tento uzel je list, což znamená, že ho zrušíme. Jak je zřejmé, strom zůstane po zrušení uvedeného listu vyvážený a není zapotřebí žádná jeho transformace.

Složitost operací

Vezměme operaci vyhledávání. Pro uzel, ve kterém hledaný prvek není, potřebujeme jednu až dvě operace srovnání. První se zjistí, zda pokračovat v hledání v levém podstromu. Pokud ne, druhou se zjistí, zda pokračovat v hledání v pravém podstromu anebo zda prvek už je nalezen. Pro vyhledání prvku potřebujeme maximálně $2 \times (h+1)$ operací srovnání, kde h je výška stromu. Připomeňme, že v kapitole pojednávající o stromech jsme pro zcela vyvážený binární strom odvodili logaritmický vztah mezi jeho výškou a počtem uzlů v něm. Dá se ukázat, že

Vyhledávání v AVL stromu má stejnou míru složitosti jako v úplně vyváženém stromu.

výška AVL stromu je maximálně 1,45 násobkem výšky úplně vyváženého binárního stromu se stejným počtem uzlů. Což znamená, že i v AVL stromu má vyhledávání logaritmickou složitost.

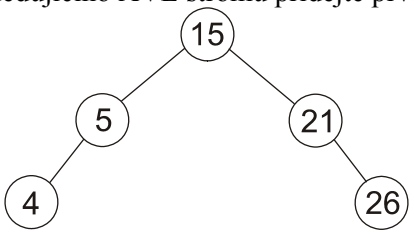
Základem zbývajících operací (přidání prvku, odebrání prvku) je vyhledání a dále průchod stromem směrem nahoru. Z čehož plyne, že složitost těchto operací rovněž závisí na výšce stromu, a to znamená, že i ony mají logaritmickou složitost.

Kontrolní otázky

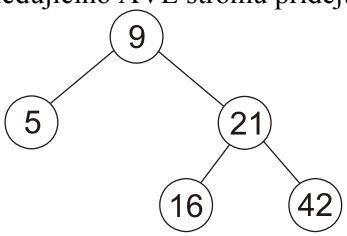
1. *Jak probíhá hledání v binárním vyhledávácím stromu?*
2. *Jaká je nejhorší možná a nejlepší možná složitost vyhledávání v binárním vyhledávacím stromu?*
3. *Proč v praxi pro binární vyhledávácí stromy nepoužíváme úplně vyvážené binární stromy, ale AVL stromy?*
4. *Jaká je časová složitost operací v AVL stromu?*
5. *Které jsou základní transformace obnovení vyvážení AVL stromu?*

Cvičení

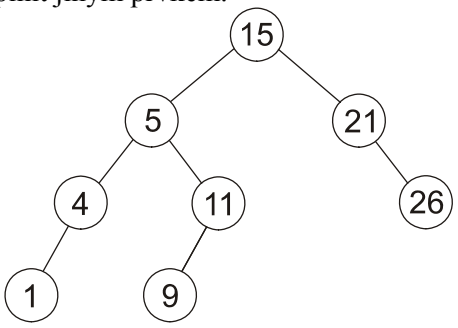
1. Do následujícího AVL stromu přidejte prvek 23.



2. Do následujícího AVL stromu přidejte prvek 18.



3. Z následujícího AVL stromu odeberte prvek 15. Ukažte všechny možnosti, jak prázdný uzel zaplnit jiným prvkem.

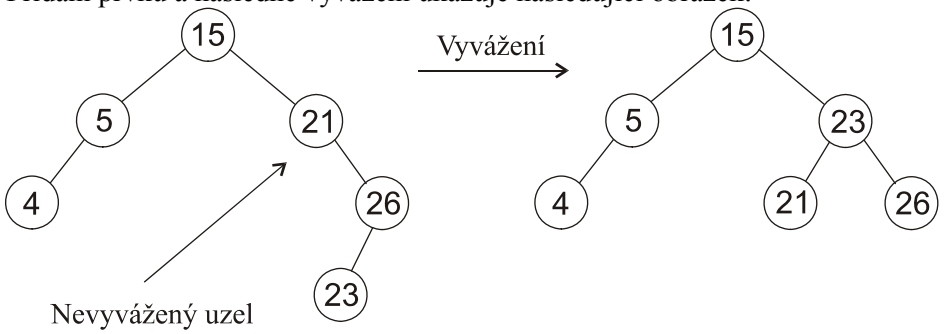


Úkoly k textu

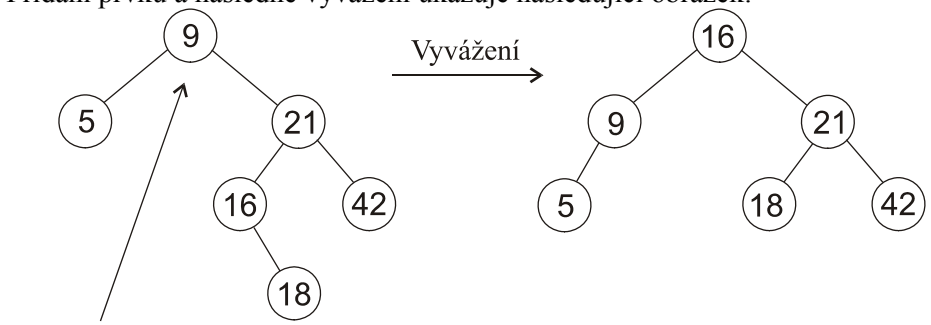
Zkuste si nakreslit AVL strom větší výšky s minimálním počtem uzlů pro danou výšku a k němu úplně vyvážený binární strom se stejným počtem uzlů a ověřte, zda skutečně pro ně platí, že poměr výšky AVL stromu a úplně vyváženého binárního stromu je nejvýše 1,45.

Řešení

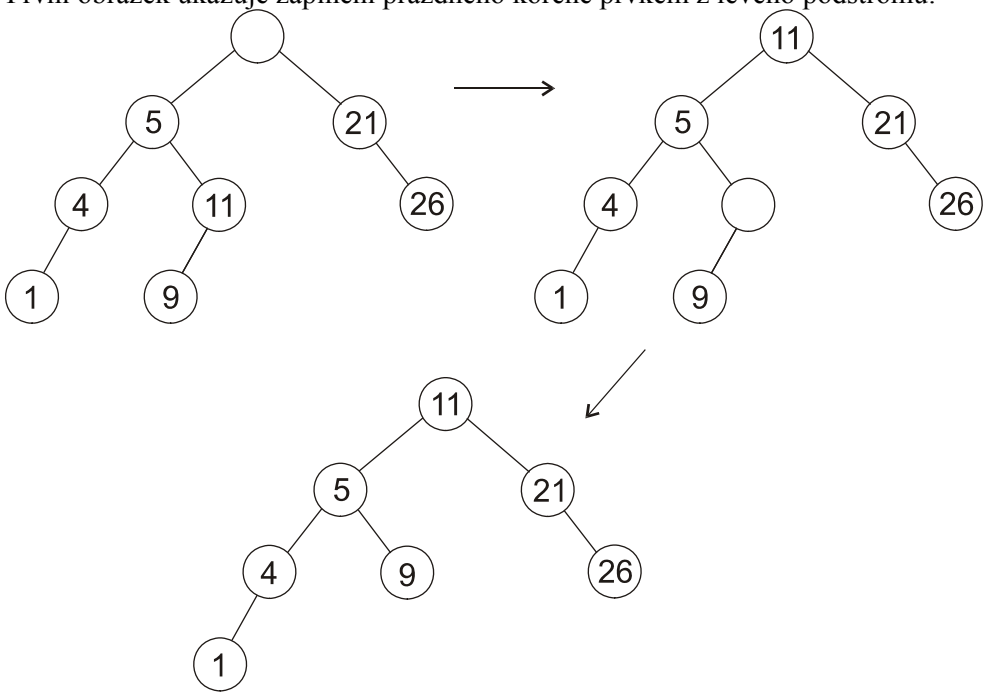
1. Přidání prvku a následné vyvážení ukazuje následující obrázek:



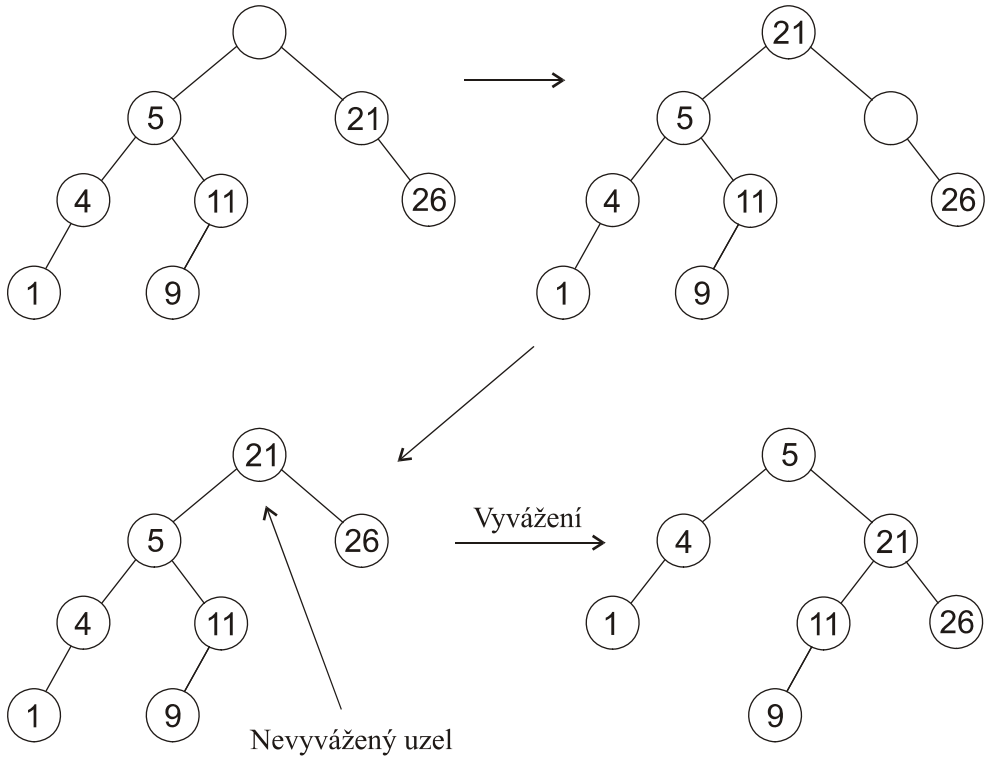
2. Přidání prvku a následné vyvážení ukazuje následující obrázek:



3. První obrázek ukazuje zaplnění prázdného kořene prvkem z levého podstromu:



Druhý obrázek ukazuje druhou možnost – zaplnění prázdného kořene prvkem z pravého podstromu:



4.3.2 B-stromy

Studijní cíle: Tato část vysvětluje princip B-stromů, popisuje jejich vlastnosti a konstrukci a ukazuje, jak v B-stromu probíhá vyhledávání, jak do B-stromu přidat prvek nebo z něho prvek odebrat a zejména, jak pak provést transformace, jimiž se zajistí zachování vyváženosti stromu.

Klíčová slova: Vložení, odebrání, B-strom.

Potřebný čas: 2 hodiny

B-stromy jsou vyhledávací stromy s více prvky v uzlu.

B-stromy jsou velmi významným typem vyhledávacích stromů. Na rozdíl od doposud uváděných stromů, kdy v každém uzlu byl uložen jeden prvek, B-stromy mají v uzlech uloženo více prvků. Struktura B-stromů je definována následujícími vlastnostmi:

Kapacita uzlu (počet prvků, který lze do uzlu uložit) je u všech uzlů stromu stejná, je to sudé číslo a volí se před začátkem vytváření stromu, označme ho *r*. Protože v B-stromech má významnou úlohu polovina z tohoto počtu, zaveďme si pro ni samostatné označení $p = \frac{r}{2}$.

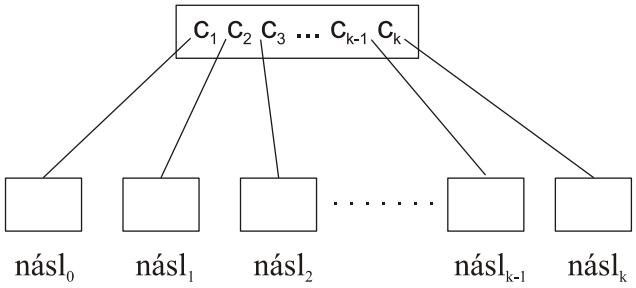
Důležité pro efektivní využití uzlů je jejich zaplnění. Všechny uzly vyjma kořene musí být aspoň z poloviny zaplněny prvky, tedy počet prvků v uzlech uložených musí být v rozmezí *p* až *r* prvků. Jedině u kořene stačí, aby obsahoval aspoň jeden prvek, tedy jeho zaplnění je v rozmezí 1 až *r* prvků.

Prvky uložené v uzlu jsou v něm seřazeny vzestupně dle velikosti.

$$c_1\ c_2\ c_3\ \ldots\ c_{k-1}\ c_k$$

$$c_1 < c_2 < c_3 < \ldots < c_{k-1} < c_k$$

Uzel je buďto list anebo má o jednoho následníka více, než je počet prvků v něm uložený.



Přitom pro prvky v jednotlivých podstromech, které těmito následníky začínají, platí:

Pro každý prvek *d* v podstromu začínajícího uzlem $násl_0$ platí $d<c_1$.

Pro každý prvek *d* v podstromu začínajícího uzlem $násl_1$ platí $c_1<d<c_2$.

Pro každý prvek *d* v podstromu začínajícího uzlem $násl_2$ platí $c_2<d<c_3$.

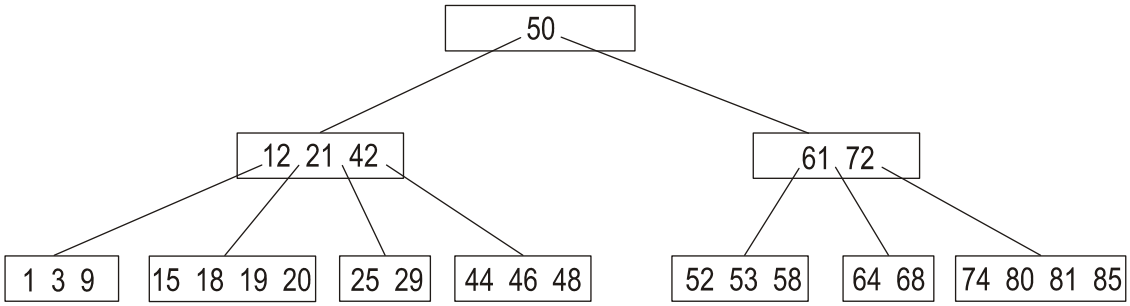
……

Pro každý prvek *d* v podstromu začínajícího uzlem $násl_{k-1}$ platí $c_{k-1}<d<c_k$.

Pro každý prvek *d* v podstromu začínajícího uzlem $násl_k$ platí $d>c_k$.

A zbývá nám poslední vlastnost, jež určuje vyváženost B-stromu. Ta stanoví, že listy jsou v B-stromu jen v jeho poslední vrstvě.

Příklad. Na následujícím obrázku je B-strom pro r=4. Prvky uložené ve stromu jsou stejně jako ve všech dosavadních příkladech celá čísla.



Postup vyhledávání

1. Počáteční krok

Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat *u*. Na začátku jím bude kořen stromu.

Hledaná hodnota nechť je *x*.

2. Průběžný krok

Provedeme vyhledání hodnoty mezi prvky uloženými v aktuálním uzlu *u*. Protože prvky jsou v uzlu setříděné, lze k tomu použít binární vyhledávání. To použijeme v případě, kdy kapacita uzlů je zvolena dostatečně velká, aby se to vyplatilo (např. $r\geq10$). Vyhledání může skončit třemi způsoby:

a) Prvek byl v aktuálním uzlu *u* nalezen, čímž vyhledávání úspěšně končí.

b) Prvek nebyl v aktuálním uzlu *u* nalezen a tento uzel je list. Tím vyhledávání končí – hledaný prvek není ve stromu obsažen.

c) Prvek nebyl v aktuálním uzlu *u* nalezen a tento uzel je nelistový. V tom případě vyhledávání skončilo v místě, kde je odkaz na následníka, ve kterém vyhledávání má pokračovat (tj. na následníka, kterým začíná podstrom, jež by hledaný prvek měl obsahovat). Tohoto následníka učiníme novým aktuální uzlem a opět se provede krok 2.

Postup přidání prvku

Chceme-li do B-stromu přidat prvek, znamená to najít příslušný uzel, do kterého nový prvek patří, a následně ověřit, zda přitom nedošlo k přeplnění, a pokud ano, provést rozdělení uzlu.

Nový prvek přidáme do listového uzlu B-stromu.

1. Přidání prvku

Označme přidávaný prvek *x*. Provedeme je vyhledání ve stromu. Použijeme k tomu běžný algoritmus pro vyhledání. Ten může skončit dvěma způsoby:

a) Prvek *x* byl ve stromu nalezen. Tím přidávání končí, neboť prvek *x* už je ve stromu obsažen (u vyhledávacích stromů se nepředpokládá vícenásobný výskyt stejného prvku).

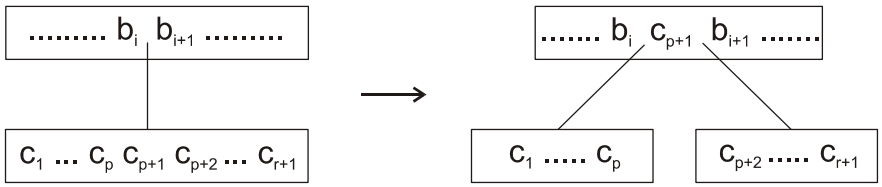
b) Vyhledávání skončilo v listovém uzlu *u* v místě, kam nový prvek podle velikosti vzhledem k ostatním prvků patří. Prvek na toto místo vložíme. Pokud uzel *u* předtím nebyl zcela zaplněn, operace přidání tím končí. Jinak provedeme rozdělení uzlu.

2. Rozdělení uzlu

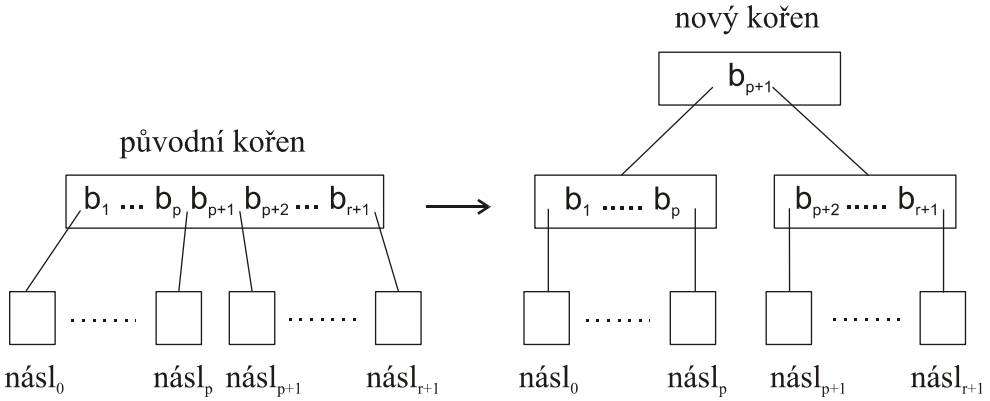
Jestliže uzel *u* má po přidání *r*+1 prvků, tedy došlo k jeho přeplnění uzlu, rozdělíme ho na tři části:

p prvků na začátku uzlu + prvek uprostřed uzlu + *p* prvků na konci uzlu.

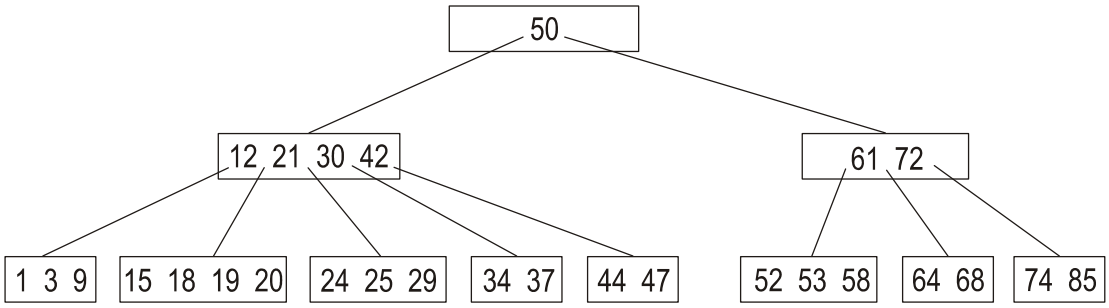
Části s *p* prvky budou tvořit nové listy. Prvek, jež je v uzlu *u* uprostřed, se přesune do předchůdce na místo, kde byl původní odkaz na list. Po přesunu nalevo a napravo od něho vytvoříme nové odkazy na nově vzniklé listy.



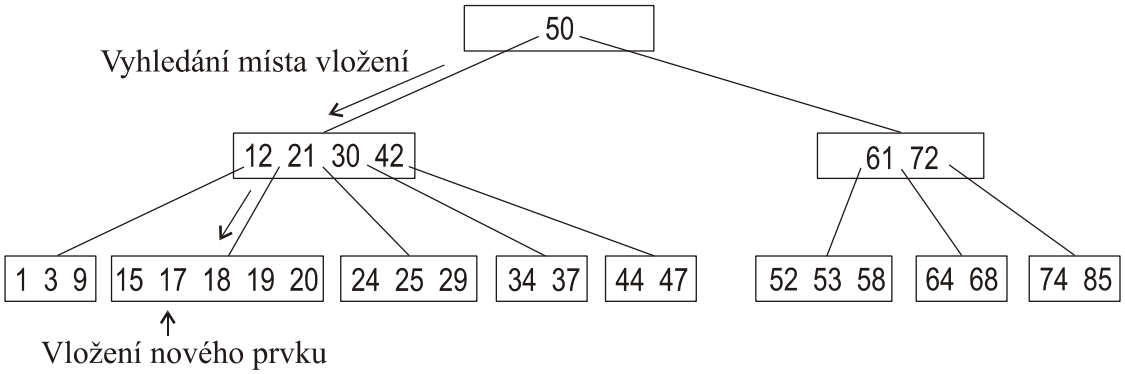
Zřejmě po přidání uzlu do předchůdce v něm může dojít rovněž k jeho přeplnění, pokud předtím byl zcela zaplněn. To se řeší stejným způsobem – rozdělením tohoto uzlu na tři části s počtem prvků *p*+1+*p*. Dvě jeho části s *p* prvky budou tvořit nové uzly a zbývající prostřední prvek vložíme do jeho předchůdce. Takto postupujeme směrem nahoru, až buďto narazíme na uzel, u kterého po vložení dalšího prvku nedojde k přeplnění, anebo se nakonec dostaneme až ke kořenu. Pokud i u něho dojde k přeplnění, rozdělí se a střední prvek v tomto rozdělení bude nyní nový kořen. V této situaci dojde ke zvětšení výšky stromu.



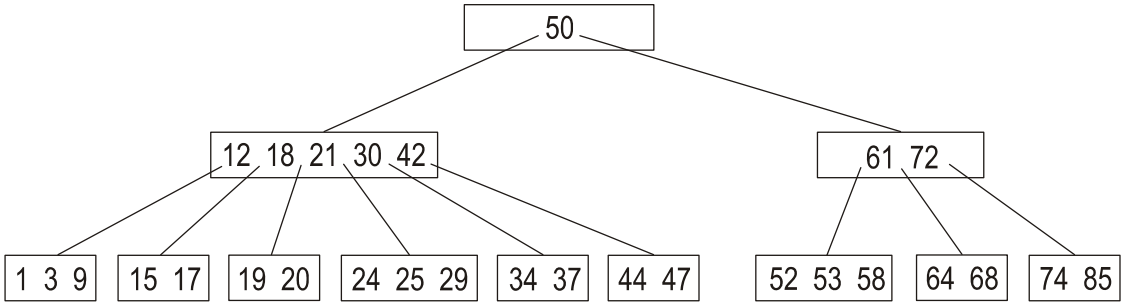
Příklad. K následujícímu B-stromu (r=4)



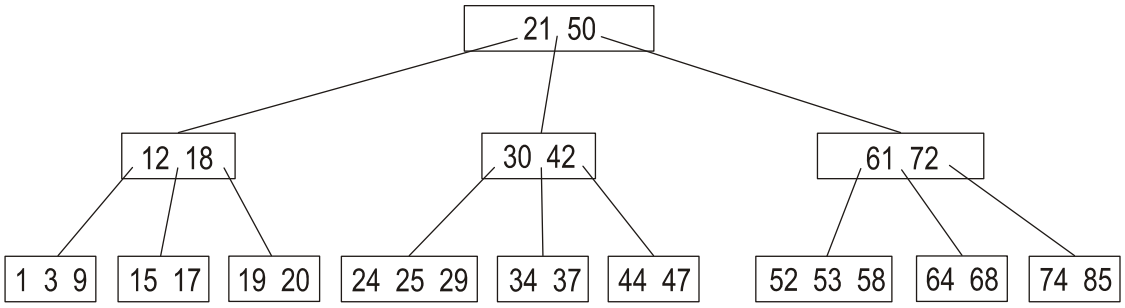
přidáme prvek 17. Následující obrázek ukazuje postup vyhledání příslušného místa, kam prvek patří, a vložení prvku.



Protože list, do kterého byl prvek 17 vložen, nyní obsahuje 5 prvků, je zapotřebí ho popsaným způsobem rozdělit.



Je zřejmé, že nyní je přeplněn předchozí uzel, do něhož byl přesunut střední prvek z rozděleného listu. Je zapotřebí rozdělit i tento uzel.



Postup odebrání prvku

Chceme-li z B-stromu odebrat prvek, znamená to vyhledat uzel, ve kterém se prvek nachází, prvek z něho odstranit a následně ověřit, zda odebráním prvku nepoklesl počet prvků v daném uzlu pod přípustnou mez, a pokud ano, je nutné to vyřešit.

Z B-stromu lze prvek snadno i odstranit.

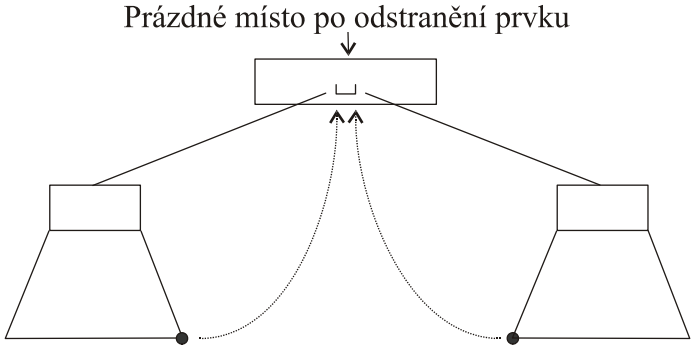
1. Odebrání prvku

Označme odstraňovaný prvek *x*. Provedeme jeho vyhledání ve stromu. Použijeme k tomu běžný algoritmus pro vyhledání. Ten může skončit třemi způsoby:

a) Prvek *x* nebyl ve stromu nalezen – není co odebrat.

b) Prvek *x* byl nalezen v listovém uzlu. Prvek z uzlu odstraníme. Pokud list je po zrušení prvku aspoň z poloviny zaplněn, operace odebrání končí. Jinak přejdeme ke kroku 2.

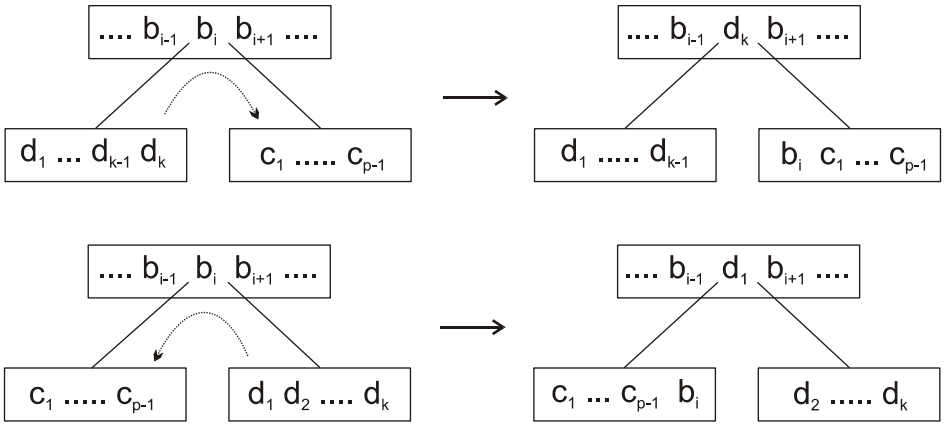
c) Prvek *x* byl nalezen v uzlu *u*, který není listem. Prvek *x* z uzlu odstraníme a na volné místo v uzlu *u* přesuneme buďto největší prvek z jeho levého podstromu, což je poslední prvek v nejpravějším listu podstromu, anebo nejmenší prvek z jeho pravého podstromu, což je první prvek v nejlevějším listu podstromu. Pokud list, odkud jsme prvek přesunuli, je stále aspoň z poloviny zaplněn, operace odebrání prvku končí. Jinak přejdeme ke kroku 2.



2. Zmenšení počtu prvků v uzlu

Sem se dostáváme v situaci, kdy po odstranění prvku ze stromu je nyní ve stromu list *v*, který má jen *p*-1 prvků. Jak se tento stav řeší, závisí na zaplnění přímých sousedů listu. Jsou dvě možnosti:

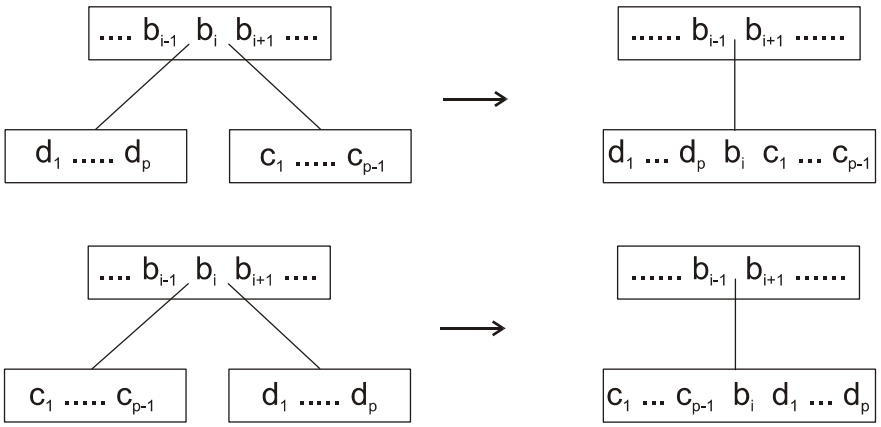
a) List *v* má aspoň jednoho přímého souseda, který má více než *p* prvků. Přímým sousedem zde máme na mysli uzel, který nejen sousedí s uzlem *v*, ale má i stejného předchůdce. Pak do listu *v* přesuneme prvek z předchůdce a na prázdné místo v předchůdci přesuneme příslušný prvek ze souseda. Následující obrázek ukazuje tento přesun pro oba možné přímé sousedy, nejdříve pro levého souseda, pak pro pravého souseda (prvku v uzlu *v* jsou značeny písmenem *c*).



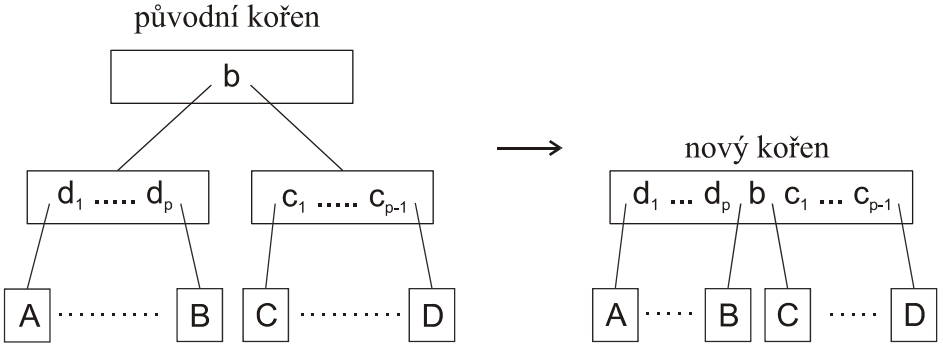
b) List v má jen přímé sousedy, které mají právě p prvků. Pak vytvoříme nový list s r prvky tak, že sloučíme

prvky z listu v + prvek z předchůdce + prvky ze souseda .

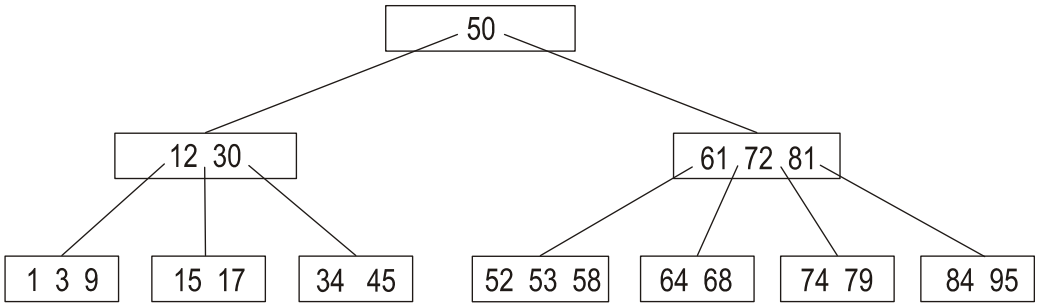
Následující obrázek ukazuje toto sloučení opět pro oba možné sousedy - levého i pravého (prvku uzlu v jsou značeny písmenem c).



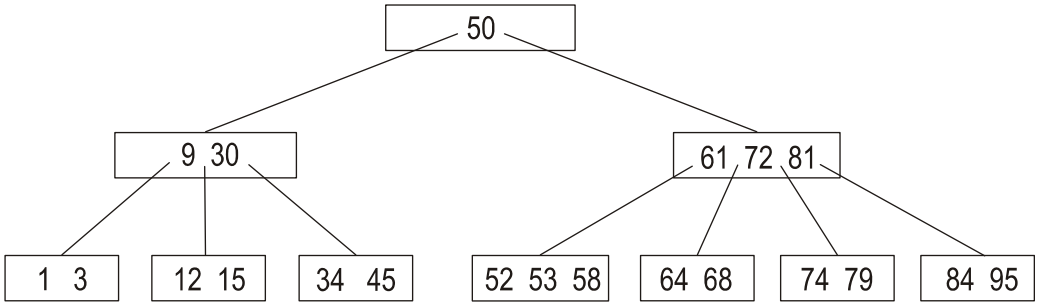
Je zřejmé, že tímto ubyl jeden prvek v předchůdci. Pokud tento má nyní jen p-1 prvků, řeší se to analogicky v závislosti na tom, kolik prvků mají jeho přímí sousedé. Takto se můžeme případně dostat až k uzlu, který je následníkem kořene. Pokud je vytvořen nový uzel sloučením s jeho přímým sousedem a pokud kořen v této chvíli má jen jeden prvek, dojde přitom k vytvoření nového kořene a zároveň ke snížení výšky stromu. Na následujícím obrázku je tato situace pro případ, kdy je pro sloučení vzat levý přímý soused.



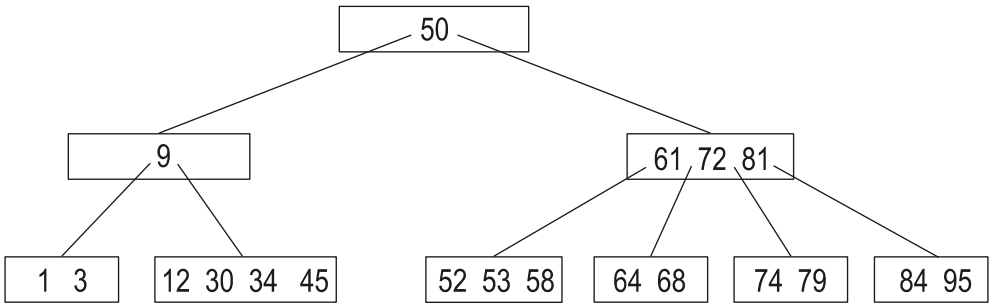
Příklad. Z následujícího B-stromu (r=4)



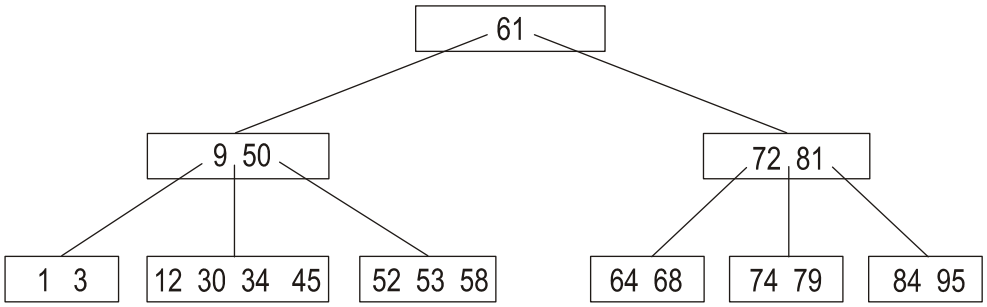
odebereme prvek 17. Po odstranění prvku z daného listu zůstane jen prvek 15. Tento list má ale levého přímého souseda, který má více než 2 prvky. Proto provedeme přesun prvku 9 z jeho levého souseda do předchůdce a prvku 12 z předchůdce do listu s prvkem 15.



Odebereme dále prvek 15. V listu zůstane opět jen jeden prvek 12. Liste už ale nemá žádného přímého souseda, z kterého by se dal přesunout prvek, proto provedeme sloučení se sousedním listem. Vybereme si k tomu třeba pravého souseda.



Při slučování byl odebrán prvek z předchůdce, ve kterém tímto zůstal jen jeden prvek 9. Tento uzel má ale pravého přímého souseda, který má více než dva prvky, čímž můžeme z něho prvek přesunout. Celý přesun se provede tak, že do uzlu s prvkem 9 se přesune prvek z jeho předchůdce (v tomto případě kořene) a na volné místo v předchůdci se přesune onen zmíněný prvek ze souseda. Přitom se příslušně přesune i ukazatel na jeho následníka – list s prvky 52,53,58.



Složitost operací

B-stromy mají dobrou časovou složitost operací.

Vezměme operaci vyhledávání. Ta jednak zahrnuje vyhledávání v uzlu. Pro ně je použito binární vyhledávání, které má logaritmickou složitost. A dále operace vyhledávání znamená procházení uzlů od kořene k listu. Protože strom je vyvážený, výška stromu logaritmicky závisí na počtu prvků ve stromu. Celkově tedy vyhledávání má logaritmickou složitost.

Základem zbývajících operací (přidání prvku, odebrání prvku) je vyhledávání a dále průchod stromem směrem nahoru. Z čehož plyne, že i tyto operace mají logaritmickou složitost.

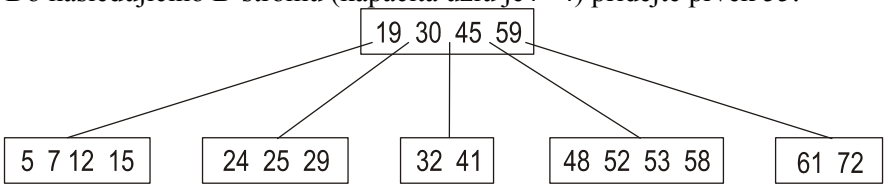
Průvodce studiem

B-stromy vznikly v roce 1969, tedy 7 let po vzniku AVL stromů, jež pocházejí z roku 1962. O B-stromech určitě ještě uslyšíte v teorii databázových systémů, neboť se v nich se hodně využívají.

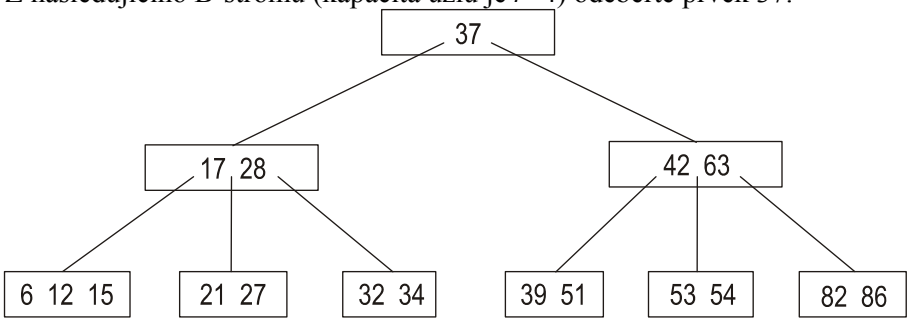
1. *Jaké jsou vlastnosti B-stromu?*
2. *Jak probíhá v B-stromu vyhledávání?*
3. *Jaká je časová složitost operací v B-stromu?*
4. *Které jsou základní transformace obnovení vyvážení B-stromu?*

Cvičení

1. Do následujícího B-stromu (kapacita uzlů je r=4) přidejte prvek 55.



2. Stejný B-strom jako v předchozím cvičení, ale nemáme přidat prvek, nýbrž máme odebrat prvek 41.
3. Z následujícího B-stromu (kapacita uzlů je r=4) odeberte prvek 37.

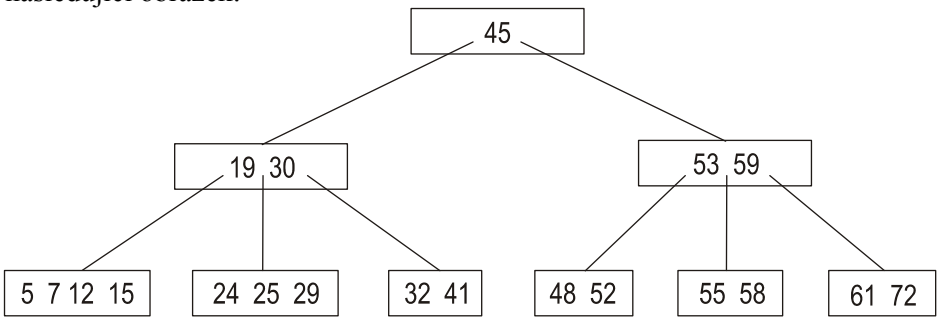


Úkoly k textu

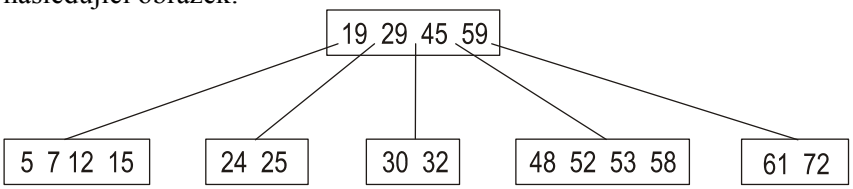
Zkuste spočítat, kolik pro danou výšku má B-strom uzlů v případě, kdy má nejmenší možné zaplnění uzlů, a kolik v případě, kdy má všechny uzly zcela zaplněny.

Řešení

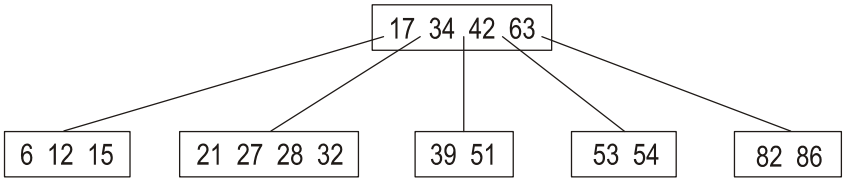
1. Po přidání prvku 55 do listu se počet prvků v listu zvýší na 5, což znamená, že je nutné z listu vytvořit dva listy a střední prvek přesunout do kořenu. Ten ale už předtím byl také plný, čímž dojde i k jeho rozdělení a vytvoření nového kořenu. Výsledek ukazuje následující obrázek:



2. Po odstranění prvku 41 z uzlu v něm zůstane jen jeden prvek. Uzel má ale přímé sousedy, které mají více než 2 prvky. Použijeme jeho levého souseda a přesuneme z něho prvek, abychom doplnili uzel, z něhož byl prvek odstraněn. Výsledek ukazuje následující obrázek:



3. Po odstranění prvku 37 z kořenu můžeme volné zaplnit největším prvkem (34) z levého podstromu anebo nejmenším prvkem (39) z pravého podstromu. Zvolme prvek 34. Ježto nyní list, který tento prvek obsahoval, má nyní jen jeden prvek a zároveň nemá přímého souseda, z kterého by bylo možné prvek do něho přesunout, sloučíme ho s jeho levým přímým sousedem. Tím ale se sníží počet prvků v předchůdci na 1 prvek, což znamená další sloučení. Výsledek ukazuje následující obrázek:



4.4 Hašování (Transformace klíče)

Studijní cíle: Tato část vysvětluje metodu hašování, popisuje konstrukci hašovací funkce a ukazuje, jak řešit kolize při stejných hodnotách hašovací funkce.

Klíčová slova: Hašování, hašovací tabulka, hašovací funkce, zřetězení.60 minut.

Potřebný čas: 1 hodina

Poslední vyhledávací metodou, kterou probereme, je hašování. Pro její anglický název *hashing* se poměrně obtížně hledá vhodný překlad do českého jazyka, proto zůstaneme u používání mírně češtině přizpůsobeného anglického názvu.

V hašování používáme k uložení prvků tabulku.

Datová struktura použitá v hašování pro uložení prvků je tabulka. Tabulka se skládá z řádků, v každém řádku je místo pro uložení jednoho prvku. Počet řádků v tabulce, tedy kapacitu tabulky označme *m*. Na jednotlivé řádky v tabulce se odkazujeme (adresujeme je) čísly 0 až *m*-

1. Při implementaci hašování se tabulka snadno realizuje pomocí pole. Datový typ se zvolí takový, aby se do něho daly uložit údaje, které ukládáme do řádků tabulky. Tím každý prvek pole reprezentuje jeden řádek tabulky a indexování pole odpovídá adresování jednotlivých řádků v tabulce.

Základem hašování je hašovací funkce. Je to zobrazení, které hodnotě prvku (nebo klíči prvku, pokud prvek je strukturovaný typ) přiřadí číslo některého z řádků v tabulce, tedy číslo v rozmezí 0 až *m*-1. Hašovací funkce se typicky sestaví ze dvou funkcí. Ta první hodnotu prvku zobrazí na celé číslo. Druhá celé číslo zobrazí na číslo řádku v tabulce, tedy na celé číslo z intervalu <0,*m*-1>. Je zřejmé, že první funkce závisí na datovém typu prvku a sestavujeme si ji sami, druhá už je víceméně standardní a jen ji použijeme.

Hašovací funkce určuje, na které místo v tabulce má být prvek uložen.

Účelem hašovací funkce je rovnoměrné rozmístění prvků v tabulce. Z toho plyne, že první funkce, která převádí hodnotu prvku na celé nezáporné číslo, by měla mít vlastnosti:

- Zobrazovat hodnoty prvků na co největší počet různých celých čísel.
- Zobrazení na celá čísla by mělo být rovnoměrné (na jednotlivá čísla by se měl zobrazovat přibližně stejný počet prvků, které chceme do hašovací tabulky uložit).

Vezměme například řetězec. Řetězec můžeme chápat jako posloupnost znaků. Jsou různé možnosti, jak tuto posloupnost zobrazit na celá čísla. Nejčastěji se při tom vychází z ASCII tabulky, která poskytuje zobrazení znaků na čísla z intervalu <0,255>. Řetězec si označme

$$z_1 z_2 ... z_k$$

kde z_i jsou jednotlivé znaky v řetězci a *k* je počet těchto znaků v řetězci (délka řetězce).

Jedna z jednodušších funkcí je

$$c_1(z_1 z_2 ... z_k) = p * asc(z_1) + q * asc(z_2) + asc(z_k) + k \quad ,$$

kde *p* a *q* jsou zvolené konstanty, nejlépe prvočísla (např. *p*=127, *q*=31), protože ty mají nejlepší předpoklady pro rovnoměrné zobrazení do celých čísel. Funkce *asc* převádí znak na jeho ASCII hodnotu.

Dokonalejší, ale na druhé straně náročnější na výpočet, je funkce

$$c_2(z_1 z_2 ... z_k) = p^{k-1} * asc(z_1) + p^{k-2} * asc(z_2) + p^{k-3} * asc(z_3) + ... + p * asc(z_{k-1}) + asc(z_k) \ ,$$

kde *p* je konstanta, opět nejlépe prvočíslo (např. *p*=31). Pro její výpočet je účelné přepsat ji do tvaru

$$c_2(z_1 z_2 ... z_k) = p * \big(...p * \big(p * \big(p * asc(z_1) + asc(z_2)\big) + asc(z_3)\big)... + asc(z_{k-1})\big) + asc(z_k)$$

Druhá část hašovací funkce, která převádí celé číslo na číslo řádku v hašovací tabulce, je velmi jednoduchá. Používá se pro ni operace *mod*, což je zbytek po celočíselném dělení. Obecný zápis hašovací funkce je

$$h(x) = c(x) \bmod m \quad ,$$

kde *c(x)* je první část hašovací funkce, která nám převádí hodnotu prvku na celé číslo, *m* je rozsah (počet řádků) hašovací tabulky. Opět je nejlepší zvolit *m* prvočíslo, protože to nemá žádného netriviálního vlastního dělitele, čímž poskytuje nejlepší předpoklady pro rovnoměrné rozmístění prvků v tabulce.

Příklad. Máme navrhnout hašování pro uložení řetězců. Velikost tabulky požadujeme přibližně 500 řádků.

Nebližší prvočísla jsou 499 a 503, velikost tabulky zvolíme třeba 503 řádků. Pro sestavení hašovací funkce použijeme již uvedenou funkci c_1. Hašovací funkce bude mít tvar:

$$h(z_1 z_2 ... z_k) = (127 * asc(z_1) + 31 * asc(z_2) + asc(z_k) + k) \bmod 503$$

Příklad. Do tabulky velikosti 11 budeme ukládat řetězce. Hašovací funkci zvolíme

$$h(z_1 z_2 ... z_k) = c(z_1 z_2 ... z_{k1}) \bmod 11 \quad ,$$

kde

$$c(z_1 z_2 ... z_k) = 7 * asc(z_1) + 3 * asc(z_2) + asc(z_k) + k \quad .$$

Do tabulky uložíme jména

h(Eva) = (7∗69+3∗118+97+3) mod 11 = 2

h(Irena) = (7∗73+3∗114+97+5) mod 11 = 9

h(Pavel) = (7∗80+3∗97+108+5) mod 11 = 7

h(Marta) = (7∗77+3∗97+97+5 mod 11 = 8

h(Ivan) = (7∗73+3∗118+110+4) mod 11 = 0

h(Nina) = (7∗78+3∗105+97+4) mod 11 = 5

Číslo řádku	Uložený prvek
0	Ivan
1	
2	Eva
3	
4	
5	Nina
6	
7	Pavel
8	Marta
9	Irena
10	

Kdybychom nyní chtěli do tabulky uložit jméno Helena, jehož hodnota hašovací funkce je

h(Helena) = (7∗72+3∗101+97+6) mod 11 = 8 ,

zjistíme, že tato pozice už je obsazená jménem Marta. Takové kolize, kdy více prvků se zobrazuje na stejný řádek hašovací tabulky, se v hašování vyskytují zcela běžně. Uvedeme si nyní nejpoužívanější způsoby jejich řešení.

4.4.1 Otevřené adresování

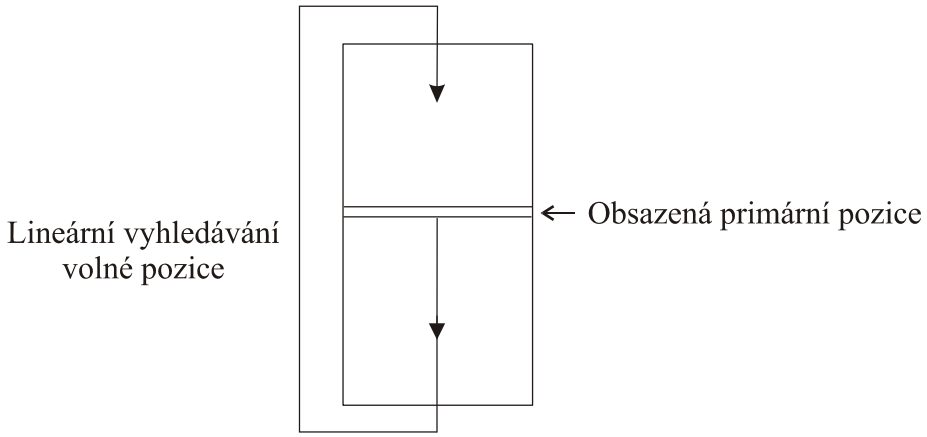
Jednou z možností řešení konfliktů je otevřené adresování.

Metoda otevřeného adresování počítá v případě, kdy pozice v tabulce vypočítaná hašovací funkcí je obsazena, další pozice tak dlouho, dokud se nenajde volná pozice anebo se nezjistí, že tabulka už je zaplněna. Nejjednodušší je lineární hledání, kdy nové pozice počítáme funkcí

$$H(x,i) = (h(x) + i) \bmod m \quad ,$$

kde *h(x)* je výchozí hašovací funkce, *i* je celočíselný parametr a *m* je rozsah tabulky. Je-li tedy primární pozice (*h(x) = H(x,0)*) obsazena, prohledávají se postupně další pozice

$$H(x,1), H(x,2), \ldots, H(x,m-1)$$



Lineární umísťování za sebou vede k vytváření nežádoucích shluků. Shluky zde označují větší počty za sebou následujících obsazených řádků tabulky. Pokud je vypočítaná primární pozice obsazená a je přitom uvnitř takového shluku, znamená to při lineárním hledání, že musíme projít všechny řádky v tomto shluku za primární pozicí, než se dostaneme k nějaké volné sekundární pozici, abychom prvek do ní mohli uložit. Přitom shluky prodlužují nejen operaci přidání prvku do tabulky, ale také vyhledávání prvku v tabulce. Při vyhledávání začínáme na primární pozici a pokud na ní prvek není a tato pozice je přitom obsazena (je na ní jiný prvek), procházíme sekundární pozice tak dlouho, dokud prvek nenajdeme nebo se nedostaneme k volné pozici, což je příznakem toho, že hledaný prvek v tabulce není.

Proto místo lineárního hledání se často používá kvadratické hledání. U něho sice také vznikají shluky, ale už v menší míře. Hašovací funkce používaná pro kvadratické hledání má běžně tvar

$$H(x,i) = (h(x) + i^2) \bmod m \ .$$

U ní je už určitý problém, že během hledání se můžeme touto funkcí dostat znovu na stejnou pozici, aniž jsme prošli celou tabulku. Nechť například hodnota hašovací funkce h pro nějaký prvek y je $h(y)=3$ a mějme tabulku s 5 řádky ($m=5$). Pak u kvadratického hledání dostáváme pozice:

$H(y,0) = (3+0^2) \bmod 5 = 3$
$H(y,1) = (3+1^2) \bmod 5 = 4$
$H(y,2) = (3+2^2) \bmod 5 = 2$
$H(y,3) = (3+3^2) \bmod 5 = 2$

Je zřejmé, že při čtvrtém pokusu (pro $i=3$) vypočítat novou pozici jsme se dostali na pozici, která už předtím byla. Dá se ale ukázat, že to nastane (za předpokladu, že m je prvočíslo), až když prohledáme nejméně polovinu tabulky. V našem příkladu to nastalo, až když jsme prohledali 3 pozice, což je více než polovina z 5. V běžném použití, pokud už tabulka není téměř zaplněna, najdeme nějakou volnou pozici většinou dříve, než projdeme polovinu tabulky.

Ještě propracovanější je metoda dvojího hašování. U ní funkce pro hledání má obecný tvar

$$H(x,i) = (h(x) + i * h_2(x)) \bmod m$$

kde $h_2(x)$ je další (sekundární) hašovací funkce, která ale nabývá jen hodnoty v rozmezí 1 až m-1 (tedy ne hodnotu 0).

V praxi se sekundární hašovací funkce $h_2(x)$ nejčastěji vytváří přímo z primární hašovací $h(x)$, která sama je sestavena z nějaké výchozí funkce $c(x)$ a má tvar

$$h(x) = c(x) \bmod m$$

Z ní použijeme její základní část, funkci $c(x)$, a hašovací funkci $h_2(x)$ vytvoříme ve tvaru

$$h_2(x) = 1 + (c(x) \bmod (m-1))$$

Funkci

$$H(x,i) = (h(x) + i * h_2(x)) \bmod m$$

upravíme na tvar

$$H(x,i) = (h(x) + (i-1) * h_2(x) + h_2(x)) \bmod m =$$
$$= ((h(x) + (i-1) * h_2(x)) \bmod m + h_2(x)) \bmod m = (H(x,i-1) + h_2(x)) \bmod m \quad .$$

Pozice vložení nyní můžeme počítat rekurzívně

$$H(x,0) = h(x)$$

$$H(x,i) = (H(x,i-1) + h_2(x)) \bmod m \quad \text{pro } i = 1,2,3,\ldots.$$

Příklad. Do tabulky vytvořené v předchozím příkladu chceme uložit další jména

h(Helena) = c(Helena) mod 11 = 8, kde c(Helena) = 7∗72+3∗101+97+6 = 910

h(Bohumil) = c(Bohumil) mod 11 = 8, kde c(Bohumil) = 7∗66+3∗111+108+7 = 910

h(Jana) = c(Jana) mod 11 = 8, kde c(Jana) = 7∗74+3∗97+97+4 = 910

U všech je hodnota hašovací funkce rovna 8, přičemž tato pozice je již obsazena. Zvolíme-li kvadratické hledání, pak další možné pozice jsou

$(8+1^2) \bmod 11 = 9$
$(8+2^2) \bmod 11 = 3$
$(8+3^2) \bmod 11 = 6$
$(8+4^2) \bmod 11 = 2$
$(8+5^2) \bmod 11 = 1$

Z vypočtených pozic jsou volné pozice 3, 6, 1. Do nich umístíme 3 nová jména:

Číslo řádku	Uložený prvek
0	Ivan
1	Jana
2	Eva
3	Helena
4	
5	Nina
6	Bohumil
7	Pavel
8	Marta
9	Irena
10	

Příklad. Do tabulky ukládáme stejná jména jako v předchozím příkladě, pro výpočet pozic použijeme dvojí hašování. Sekundární hašovací funkci bude

$$h_2(x) = 1 + (c(x) \bmod 10) \quad .$$

Její hodnota pro ukládaná jména je

h_2(Helena) = h_2(Bohumil) = h_2(Jana) = 1 + 910 mod 10 = 1

Pro vkládaná jména je hodnota primární funkce rovna 8. Z ní vypočítáme hodnotu sekundární hašovací funkce

h_2(Helena) = h_2(Bohumil) = h_2(Jana) = 1+(8 mod 5) = 4

Sekundární pozice budou

(8+1) mod 11 = 9

(9+1) mod 11 = 10

(10+1) mod 11 = 0

(0+1) mod 11 = 1

(1+1) mod 11 = 2

(2+1) mod 11 = 3

Z vypočtených pozic jsou volné pozice 10, 1, 3. Do nich umístíme 3 nová jména:

Číslo řádku	Uložený prvek
0	Ivan
1	Bohumil
2	Eva
3	Jana
4	
5	Nina
6	
7	Pavel
8	Marta
9	Irena
10	Helena

Vyhledávání v tabulce

Při vyhledávání v hašovací tabulce nejprve vypočítáme hodnotu hašovací funkce pro hledaný prvek *x*. Podíváme se do tabulky na řádek, na který ukazuje hodnota hašovací funkce. Mohou nastat případy:

a) Řádek je prázdný – hledaný prvek není v tabulce.

b) Na řádku je hledaný prvek *x* – vyhledávání tím úspěšně končí.

c) Na řádku je jiný prvek než *x*. Začneme postupně počítat další možné pozice a srovnávat prvky na nich s hledaným prvkem *x*, dokud buďto hledaný prvek nenalezneme anebo se nedostaneme na prázdný řádek anebo nevyčerpáme všechny možné pozice.

Příklad. Máme vyhledat jméno Robert v tabulce z předminulého příkladu.

h(Robert)=(7*82+3*111+116+6) mod 11 = 6

Na této pozici je ale jiné jméno - Bohumil. Začneme počítat a prohledávat další možné pozice

(6+1^2) mod 11 = 7 - Pavel

$(6+2^2) \bmod 11 = 10$

Vyhledávání skončí na pozici 10, která je prázdná. Jméno Robert tedy v tabulce není.

4.4.2 Zřetězení

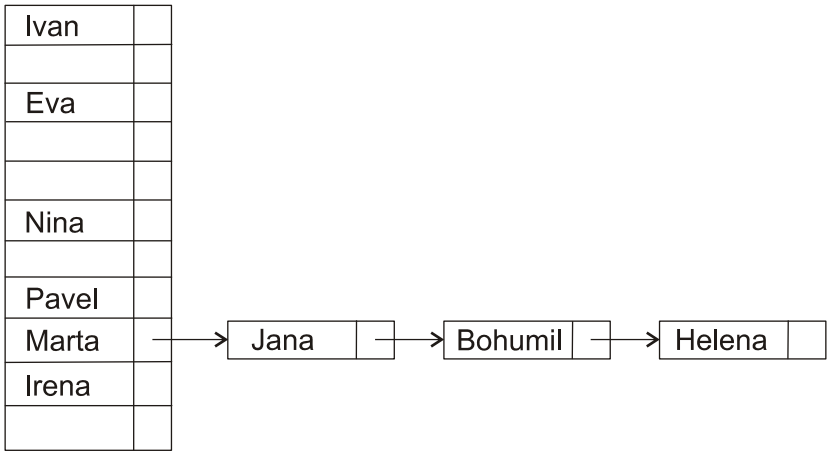
Předchozí metoda otevřeného adresování má dvě nevýhody:

- Počet prvků, jež lze do tabulky uložit, je omezen její velikostí. Pokud dopředu neznáme, kolik prvků bude do tabulky ukládáno, může se stát, že ji stanovíme malou a dojde k jejímu přeplnění. Následné zvětšení velikosti tabulky může být časově náročně.
- Při vyhledávání, zejména v dost zaplněné tabulce, procházíme v důsledku otevřeného adresování i prvky, které mají jinou hodnotu hašovací funkce, čímž se doba vyhledávání zvětšuje.

Další možnost řešení konfliktů je metoda zřetězení, která využívá seznamy.

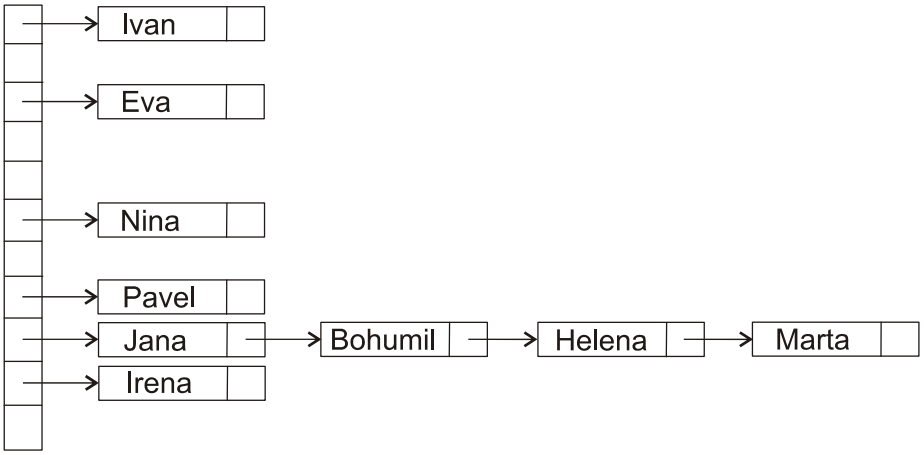
Tyto nevýhody odstraňuje metoda zřetězení, která k ukládání dalších prvků se stejnou hodnotou hašovací funkce využívá seznamy. Řádek v hašovací tabulce v tomto případě má kromě místa na uložení prvku dále místo pro ukazatel na seznam.

Příklad. Tabulka z předchozích příkladů, ale se zřetězením.



Nebo druhá možnost je, že hašovací tabulka je tvořena jen ukazateli a všechny prvky jsou uloženy v seznamech. Zejména je tento způsob výhodný v případě, kdy do hašovací tabulky nejen přidáváme nové prvky, ale rovněž z ní prvky i odebíráme.

Příklad. Tabulka z předchozího příkladu se zřetězením. Všechny prvky jsou nyní uloženy v seznamech.



Složitost hašování

Při vyhledávání je složitost dána složitostí výpočtu hašovací funkce a složitostí vlastního vyhledávání. Složitost hašovací funkce závisí na tom, jak se počítá. Nicméně ji můžeme považovat za konstantní. Například u řetězců pracujeme buďto s řetězci, které mají omezenou délku, anebo případně můžeme pro výpočet hašovací funkce brát jen omezený počet znaků z řetězce, například uvažujeme jen jeho 20 znaků.

U hašování se dá dosáhnout vynikající časová složitost vyhledávání.

Nejlepší situace v hašování je, když při ukládání prvků (vytváření hašovací tabulky) nedojde k žádné kolizi, pak složitost vyhledávání je viditelně $\Theta(1)$, což neposkytuje žádná dosud popsaná vyhledávací metoda. Běžně ale kolize nastávají. Složitost přirozeně roste s počtem kolizí v tabulce. Budeme-li uvažovat metodu zřetězení pro řešení konfliktů, pak pro vyhledání prvku potřebujeme počet srovnání, který se pohybuje od 1 (když prvek je okamžitě nalezen) až po maximum z délek seznamů (když prohledáváme seznam s dalšími prvky se stejnou hodnotou hašovací funkce). Věcně vzato, jestliže zvolíme dostatečně velkou hašovací tabulku vzhledem k očekávanému množství ukládaných prvků (tj. o něco větší) a zvolíme dostatečně kvalitní hašovací funkci, která rovnoměrně zobrazuje prvky do hašovací tabulky, lze očekávat že počet srovnání potřebný pro vyhledání prvku bude typicky velmi nízký, tedy jen několik srovnání.

Volba hašovací tabulky je zejména podstatná u metody otevřeného adresování. Volíme ji aspoň o 10% větší, než je očekávaný počet prvků. Empirické testy ukazují, že při 90% zaplnění tabulky je i při nejjednodušší metodě řešení kolizí, kterou je lineární hledání, v průměru zapotřebí 5,5 srovnání pro nalezení libovolného prvku. U promyšlenějších metod (kvadratické hledání, dvojí hašování) je průměrný počet pokusů ještě menší.

Problémem tabulek s otevřeným adresováním je případ, kdy nelze předvídat počet prvků, které do ní budou uloženy, a nelze tedy vyloučit situaci, že tabulka se zaplní do takové míry, že už se nenajdou volné pozice pro uložení dalších prvků. Pokud by toto mohlo nastat, je možnost tento problém řešit tak, že vytvoříme novou, přiměřeně větší hašovací tabulku, prvky z dosavadní tabulky do ní přesuneme a předchozí tabulku zrušíme (uvolníme paměť, která pro ni byla vyhrazena).

Pokud shrneme, co jsme o hašovacích tabulkách doposud uvedli, je zřejmé, že hašování je velmi účinná a efektivní metoda vyhledávání a proto jsou hašovací tabulky velmi často používané. Navíc vlastní algoritmus a tím i implementace hašování je poměrně jednoduchý, mnohem jednodušší než třeba vyhledávací stromy.

Odebírání prvků z hašovací tabulky

Zatím jsme se zabývali jen operací přidání prvku. V řadě použití hašovacích tabulek to postačuje. Pokud bychom potřebovali i odebírat prvky, pak je to snadné v tabulkách, které používají metodu zřetězení a jsou přitom tvořeny jen ukazateli a mají všechny prvky uloženy v seznamech. Jde o tabulky popsané na konci části, v níž byla popsána metoda zřetězení. Zde odebírání prvku je realizováno odebíráním prvku ze seznamu.

U tabulek založených na otevřeném adresování se odebrání prvku řeší tím způsobem, že řádek se po odebrání prvku označí specifickou hodnotou. Toto označení ho odlišuje od prázdných řádků, což jsou řádky, do kterých nebyl ještě uložen žádný prvek. Řádek takto označený je v operaci přidávání použit stejným způsobem pro uložení nového prvku jako prázdný řádek. V operaci vyhledávání se ale tento řádek jen přeskočí a vyhledávání se na něm nezastaví jako u prázdného řádku.

Algoritmy hašování jsou velmi používané. Třeba překladač (kompilátor) je využívá pro uložení informací o překládaném programu. Využívají se rovněž v dabázových systémech atd.

Kontrolní otázky

1. *Jaké sestavíme hašovací funkci?*

2. *Čím při hašování vznikají kolize a jak je řešíme?*
3. *Jaká je časová složitost hašování?*

Cvičení

1. Sestavte hašovací tabulku, do které budou ukládány řetězce. Abychom zjednodušili výpočet hašovací funkce, budeme předpokládat, že řetězce budou začínat jen velkými písmeny. Pro převod písmen na čísla použijeme velmi jednoduchou funkci, kterou nazveme *w* a která písmena zobrazí na celá čísla podle předpisu
   w(A)=1, w(B)=2, w(C)=3,… w(Y)=25, w(Z)=26 .
   Pokud písmeno má diakritiku, tak ji zanedbáme.
   Rozsah tabulky bude 7 řádků a hašovací funkce bude velmi jednoduchá
   $$h(z_1 z_2 ... z_k) = (w(z_1) + k) \bmod 7$$
   Uložte do tabulky jména Marta, Petr, Irena.
2. Budeme pokračovat v předchozím cvičení a pro řešení konfliktů zvolíme otevřené adresování a použijeme dvojí hašování. Další (druhou) hašovací funkce sestavte podle první funkce (uvedené v předchozím cvičení). Pak do tabulky uložte jméno Jiří.

Úkoly k textu

Máte vytvořit hašovací tabulku s určitými záznamy o osobách. Součástí záznamů jsou rodná čísla, která slouží jako klíč. Zkuste se zamyslet, jak byste z rodného čísla vytvořili hašovací funkci.

Řešení

1. Hodnoty hašovací funkce jsou
   h(Marta) = (13+5) mod 7 = 4
   h(Petr) = (16+4) mod 7 = 6
   h(Irena) = (9+5) mod 7 = 0
   Hašovací tabulka je:

0	Irena
1	
2	
3	
4	Marta
5	
6	Petr

2. Hašovací funkce pro řešení konfliktů je:
   $$H(z_1 z_2 ... z_k, i) = (w(z_1) + k + i * (1 + (w(z_1) + k) \bmod 6)) \bmod 7$$
   Vypočítáme hodnotu hašovací funkce pro jméno Jiří
   h(Jiří) = (10+4) mod 7 = 0
   Pozice 0 je v tabulce již obsazena jménem Irena. Vypočítáme další pozici
   H(Jiří,1) = (10+4+1∗(1+(10+4) mod 6)) mod 7 = 3

Ta je již volná a jméno Jiří lze na ni uložit:

0	Irena
1	
2	
3	Jiří
4	Marta
5	
6	Petr

5 Rejstřík