ALG

11. Základní algoritmy a datové struktury pro vyhledávání a řazení. Vyhledávací stromy, rozptylovací tabulky. Prohledávání grafu. Úlohy dynamického programování. Asymptotická složitost a její určování.

Řád růstu funkcí, asymptotická složitost algoritmu.

Asymptotická složitost je způsob klasifikace počítačových algoritmů. Určuje operační náročnost algoritmu tak, že zjišťuje jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti (počtu) vstupních dat.

Třída složitosti

Funkce vyjadřující počet operací potřebných ke zpracování dat.
$1 \ll log(n) \ll n \ll n \cdot log(n) \ll n^k \ll k^n \ll n! \ll n^n$
Pokud spadají dva algoritmy do různých tříd asymptotické složitosti, pak vždy existuje takové množství dat, od kterého je asymptoticky lepší algoritmus vždy rychlejší, bez ohledu na to, kolikrát je některý z počítačů výkonnější.

Řád růstu funkcí

U většiny algoritmů nelze říci, že jejich složitost odpovídá přesně jedné třídě, protože rychlost algoritmu závisí také na povaze dat. Z tohoto důvodu se používáme řád růstu funkcí, který zohledňuje nejhorší i nejlepší možný běh algoritmu.

- $O(f(x))$ - $Omicron(f(x))$ – algoritmus probíhá asymptoticky stejně rychle nebo pomaleji než $f(x)$, horní odhad
- $\Omega(f(x))$ - $Omega(f(x))$ – algoritmus probíhá asymptoticky stejně rychle nebo rychleji než $f(x)$, spodní odhad
- $\Theta(f(x))$ - $Theta(f(x))$ – algoritmus probíhá asymptoticky stejně rychle jako $f(x)$, zároveň platí $O(f(x))$ a $\Omega(f(x))$

Funkce f(x) ohraničuje funkci našeho algoritmu g(x) ,O(f(x)) vyjadřuje ohraničení shora, zatímco Ω(f(x)) zdola. Θ(f(x)) pak značí ohraničení z obou stran.

Common orders of growth.

NAME	NOTATION	EXAMPLE	CODE FRAGMENT
Constant	$O(1)$	array access arithmetic operation function call	`op();`
Logarithmic	$O(\log n)$	binary search in a sorted array insert in a binary heap search in a red–black tree	`for (int i = 1; i <= n; i = 2*i)` `  op();`
Linear	$O(n)$	sequential search grade-school addition BFPRT median finding	`for (int i = 0; i < n; i++)` `  op();`

Linearithmic	$O(n \log n)$	mergesort
		heapsort
		fast Fourier transform
			`for (int i = 1; i <= n; i++)`
			`  for (int j = i; j <= n; j = 2*j)`
			`  op();`

Quadratic	$O(n^2)$	enumerate all pairs
		insertion sort
		grade-school multiplication
			`for (int i = 0; i < n; i++)`
			`  for (int j = i+1; j < n; j++)`
			`  op();`

Cubic	$O(n^3)$	enumerate all triples
		Floyd–Warshall
		grade-school matrix multiplication
			`for (int i = 0; i < n; i++)`
			`  for (int j = i+1; j < n; j++)`
			`  for (int k = j+1; k < n; k++)`
			`  op();`

Polynomial	$O(n^c)$	ellipsoid algorithm for LP AKS primality algorithm Edmond's matching algorithm	
Exponential	$2^{O(n^c)}$	enumerating all subsets enumerating all permutations backtracing search	

Properties of order-of-growth notations.

- *Reflexivity:* $f(n)$ is $O(f(n))$.
- *Constants:* If $f(n)$ is $O(g(n))$ and $c > 0$, then $c \cdot f(n)$ is $O(g(n))$.
- *Products:* If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) \cdot f_2(n)$ is $O(g_1(n) \cdot g_2(n))$.
- *Sums:* If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max\{g_1(n), g_2(n)\})$.
- *Transitivity:* If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.
- *Polynomials:* Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.
- *Logarithms and polynomials:* $\log_b n$ is $O(n^d)$ for every $b > 0$ and every $d > 0$.
- *Exponentials and polynomials:* n^d is $O(r^n)$ for every $r > 0$ and every $d > 0$.
- *Factorials:* $n!$ is $2^{\Theta(n \log n)}$.
- *Limits:* If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$, then $f(n)$ is $\Theta(g(n))$.
- *Limits:* If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Theta(g(n))$.
- *Limits:* If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$, then $f(n)$ is $\Omega(g(n))$ but not $O(g(n))$.

Here are some examples.

FUNCTION	$o(n^2)$	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	$\omega(n^2)$	$\sim 2n^2$	$\sim 4n^2$
$\log_2 n$	✔	✔					
$10n + 45$	✔	✔					
$2n^2 + 45n + 12$		✔	✔	✔		✔	
$4n^2 - 2\sqrt{n}$		✔	✔	✔			✔
$3n^3$				✔	✔		
2^n				✔	✔		

Divide-and-conquer recurrences. For each of the following recurrences we assume $T(1) = 0$ and that $n / 2$ means either $\lfloor n / 2 \rfloor$ or $\lceil n / 2 \rceil$.

RECURRENCE	$T(n)$	EXAMPLE
$T(n) = T(n / 2) + 1$	$\sim \lg n$	*binary search*
$T(n) = 2T(n / 2) + n$	$\sim n \lg n$	*mergesort*
$T(n) = T(n - 1) + n$	$\sim \frac{1}{2} n^2$	*insertion sort*
$T(n) = 2T(n / 2) + 1$	$\sim n$	*tree traversal*
$T(n) = 2T(n - 1) + 1$	$\sim 2^n$	*towers of Hanoi*
$T(n) = 3T(n / 2) + \Theta(n)$	$\Theta(n^{\log_2 3}) = \Theta(n^{1.58\ldots})$	*Karatsuba multiplication*
$T(n) = 7T(n / 2) + \Theta(n^2)$	$\Theta(n^{\log_2 7}) = \Theta(n^{2.81\ldots})$	*Strassen multiplication*
$T(n) = 2T(n / 2) + \Theta(n \log n)$	$\Theta(n \log^2 n)$	*closest pair*

Master theorem. Let $a \geq 1$, $b \geq 2$, and $c > 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the divide-and-conquer recurrence

$$T(n) = a\, T(n / b) + \Theta(n^c)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n / b means either $\lfloor n / b \rfloor$ or either $\lceil n / b \rceil$.

- If $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$
- If $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$
- If $c > \log_b a$, then $T(n) = \Theta(n^c)$

Remark: there are many different versions of the master theorem. The Akra–Bazzi theorem is among the most powerful.

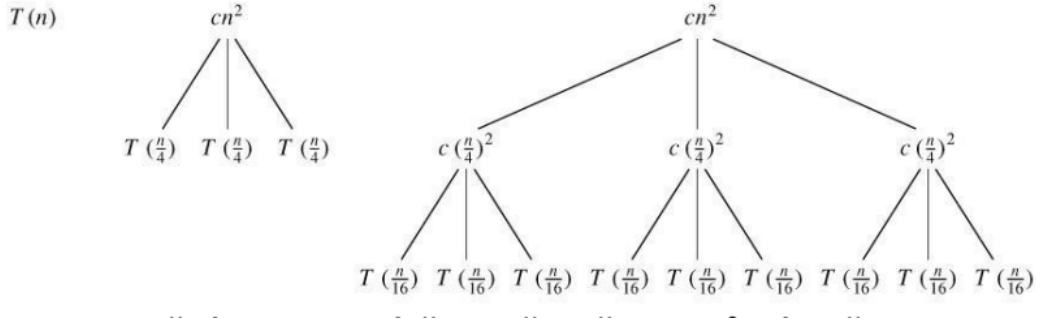
Rekurze, Stromy, Binární stromy, prohledávání s návratem

Rozděl a panuj

Tato metoda označuje ty algoritmy pro práci s daty, které řeší problém rozdělením řešené úlohy na **dílčí části (podproblémy)**, nad kterými se provádí algoritmická operace. Často se tato metoda implementuje rekurzivně nebo iterativně a původní úloha se dělí na stále menší části.

Rekurze

Iterativně se úloha rozkládá do rekurzivních stromů (grafická vizualizace rozvoje rekurze).



Mistrovská věta

$T(n) = aT(n/b) + f(n)$, kde $a >= 1$, $b > 1$ a f(n) je asymptoticky kladná funkce.

1. Pokud $f(n) \in O(n^{\log_b(a)-\varepsilon})$ pro nějakou konstantu $\varepsilon > 0$, potom
$$T(n) \in \Theta(n^{\log_b(a)}).$$

2. Pokud $f(n) \in \Theta(n^{\log_b(a)})$, potom
$$T(n) \in \Theta(n^{\log_b(a)} \log(n)).$$

3. Pokud $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ pro nějakou konstantu $\varepsilon > 0$ a pokud $a f(n/b) \leq c f(n)$ pro nějakou konstantu $c < 1$ a všechna dostatečně velké n, potom
$$T(n) \in \Theta(f(n)).$$

- Příklad 1 (Strassenův algoritmus):

  $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$

- Z toho dostáváme, že $a = 7$, $b = 2$, $f(n) \in O(n^{\log_2(7)-0.5})$.
  Jedná se tedy o případ číslo 1.
- Dostáváme složitost:

  $T(n) \in \Theta(n^{\log_2(7)}) \subseteq \Theta(n^{2.8074})$

- Příklad 2 (Nákup a prodej akcií):

  $T(n) = 2 \cdot T(n/2) + \Theta(n)$

- Z toho dostáváme, že $a = 2$, $b = 2$,

  $f(n) \in \Theta(n) = \Theta(n^{\log_2(2)})$.

  Jedná se tedy o případ číslo 2.
- Dostáváme složitost:

  $T(n) \in \Theta(n \cdot \log(n))$

Stromy

Jedná se o hierarchickou strukturu, kde každý rodič má 0 až mnoho dětí a každé dítě právě jednoho rodiče takovým způsobem, že v této struktuře nejsou cykly. Uzel, který je prarodičem všech ostatních uzlů nazveme kořenem (z pohledu teorie grafů tím vytvoříme orientovaný strom). Uzel, který nemá žádné potomky nazýváme listem.
Být stromem je rekurzívní vlastnost - každý podstrom stromu S je také stromem.

Lze reprezentovat aritmetický výraz, datovou strukturu (BVS, halda), rozhodovací strom, hierarchii (rodokmen, zaměstnanci), rekurzivní volání, stavový prostor, atd...

Hloubka stromu = hloubce nejhlubšího listu, přičemž kořen má hloubku 0.

Binární strom

Binární strom je orientovaný graf s jedním vrcholem (kořenem), z něhož existuje cesta do všech vrcholů grafu. Každý vrchol binárního stromu může mít maximálně dva orientované syny a s výjimkou kořene právě jednoho předka. Kořen předka nemá.

V praktickém programování je obvykle binární strom reprezentován dvěma způsoby:

1. pomocí dynamické struktury, kde jsou hrany reprezentovány ukazateli

2. pomocí pole, kde prvek s indexem i má následníky s indexem 2i+1 a 2i+2 (za předpokladu, že pole je indexováno od 0). Takto je například reprezentovaná halda v algoritmu heapsort. (také viz. PRP)

Za zmínku stojí speciální typ binárního stromu - **Vyvážený binární** strom, jehož hloubka listů se od sebe liší maximálně o jedna.

Prohledávání s návratem

Začne se v kořeni a pro aktuální uzel se zpracuje a projde se levý a pravý podstrom.

Jsou 3 způsoby procházení a zpracování stromu:

- **PreOrder** - uzel -> levý podstrom -> pravý podstrom
- **InOrder** - levý podstrom -> uzel -> pravý podstrom
- **PostOrder** - levý podstrom -> pravý podstrom -> uzel

Příklad osmi dam na šachovnici - vytvoří se strom testovaných pozic, kde každé dítě má omezenější možnosti kvůli položení dány, velké zrychlení oproti Brute Force

Fronta, zásobník, průchod stromem/grafem do šířky/hloubky.

Fronta

- FIFO - first in first out, odebírá se z čela fronty a přidává na konec
- Lze implementovat v cyklickém poli

Zásobník

- LIFO - last in first out, odebírá se poslední vložený prvek (čelo) a vkládá se na čelo
- Lze implementovat jednoduše v poli

BFS (průchod do šířky) stromu

Vytvoří se prázdná fronta, vloží se tam kořen a dokud není fronta prázdná tak se:

- Odebere první uzel z fronty a zpracuje se
- Po zpracování se do fronty vloží jeho potomci
- Repeat

Lze rozšířit na obecné ne/orientované grafy, s tím, že máme matici sousednosti nebo seznam sousedů pro každý vrchol. Složitost $O(|V| + |E|)$.

DFS (průchod do hloubky)

Prohledává strom nebo graf do hloubky. Lze implementovat rekurzivně, nebo zásobníkem (jiné pořadí uzlů a uzel může být vložen opakovaně). Složitost $O(|V| + |E|)$.

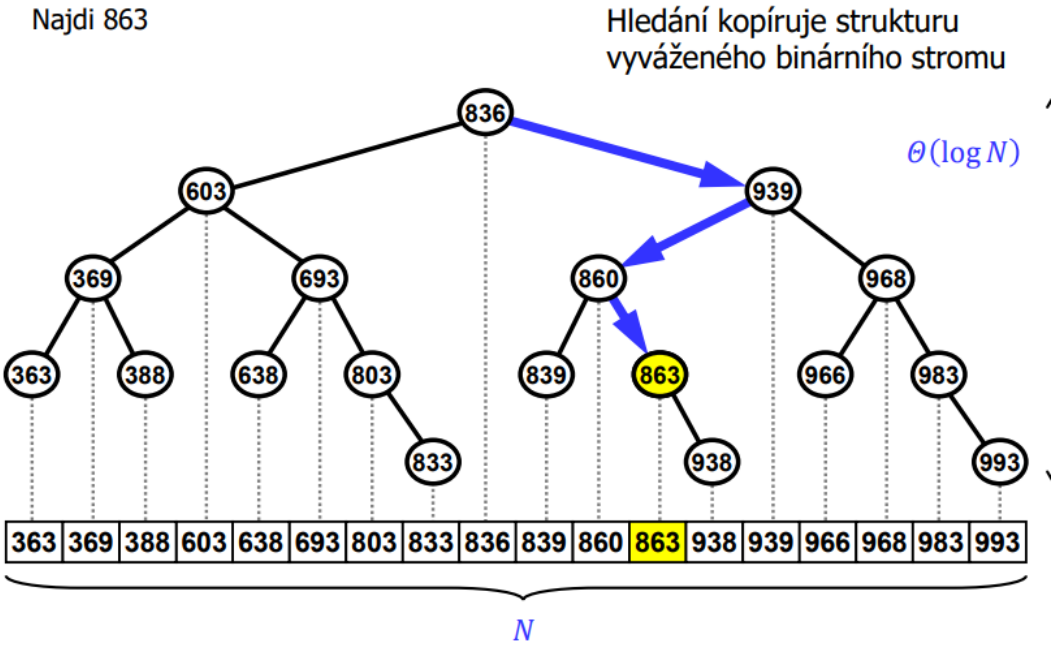
Binární vyhledávací stromy, AVL a B- stromy

Binární vyhledávací strom (BVS/BST)

Binární vyhledávací strom (BST - z angl. Binary Search Tree) je datová struktura založená na binárním stromu, v němž jsou jednotlivé prvky (uzly) uspořádány tak, aby v tomto stromu bylo možné rychle vyhledávat danou hodnotu. To zajišťují tyto vlastnosti:

- Jedná se o binární strom, každý uzel tedy má nanejvýš dva syny − levého a pravého.
- Každému uzlu je přiřazen určitý klíč. Podle hodnot těchto klíčů jsou uzly uspořádány.
- Levý podstrom uzlu obsahuje pouze klíče menší než je klíč tohoto uzlu.
- Pravý podstrom uzlu obsahuje pouze klíče větší než je klíč tohoto uzlu.

1. **Vyhledávání**
   Vyhledání konkrétní hodnoty v binárním vyhledávacím stromu typicky probíhá rekurzivně. Začíná v kořeni. V každém kroku porovná hledanou hodnotu s klíčem zkoumaného uzlu. Pokud jsou si rovny, hodnota byla nalezena. Je-li hledaná hodnota menší, pokračuje hledání v levém podstromu. Je-li větší, bude hledání pokračovat v pravém podstromu. Díky uspořádání stromu je cesta k hledané hodnotě jednoznačně určena. Časová složitost je $\theta(logN)$. Možné vylepšení pomocí exponenciálního vyhledávání dokud hodnotu nepřestřelíme a pak lineárně. Podobně jako při určení maximální rychlosti stahování dat z internetu.

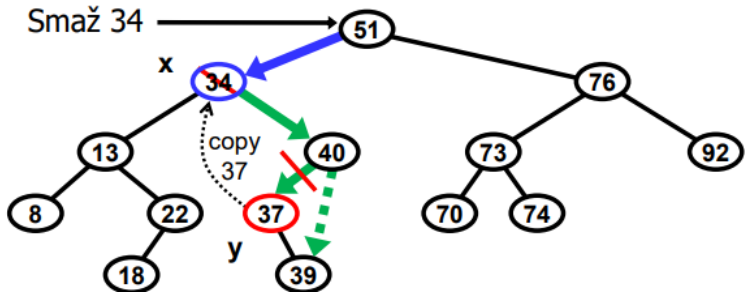


2. **Přidání uzlu**
   Vložení nového uzlu začíná hledáním jeho pozice ve stromu – postupuje se stejně jako při vyhledávání, jako hledaná hodnota se použije klíč vkládaného uzlu. Tato fáze může vést ke dvěma různým výsledkům:

- klíč byl nalezen, strom tedy dotyčnou hodnotu již obsahuje a není třeba ji vkládat (komplikovanější varianty připouštějící vícenásobný výskyt stejného klíče by pokračovaly dál do podstromu připouštějícího rovnost)
- algoritmus narazil na neexistující uzel, nový uzel bude vložen na toto místo, protože sem podle hodnoty svého klíče patří

3. **Odstranění uzlu**

- **Odstranění listu:** Odstranění uzlu bez potomků se jednoduše provede odstraněním uzlu ze stromu.
- **Odstranění uzlu s jedním potomkem:** Provede se nahrazením uzlu uzlem potomka.
- **Odstranění uzlu se dvěma potomky:** Nechť se odstraněný uzel nazývá N. Pak je hodnota uzlu N nahrazena nejbližší vyšší (uzel pravého podstromu, který je nejvíc vlevo (nejlevější)), nebo nižší hodnotou (uzel levého podstromu, který je nevíc vpravo (nejpravější)). Takový uzel má nanejvýš jednoho potomka, lze jej tedy ze stromu vyjmout podle jednoho z předchozích pravidel. V dobré implementaci je doporučeno obě varianty střídat, jinak dochází k narušení rovnováhy.

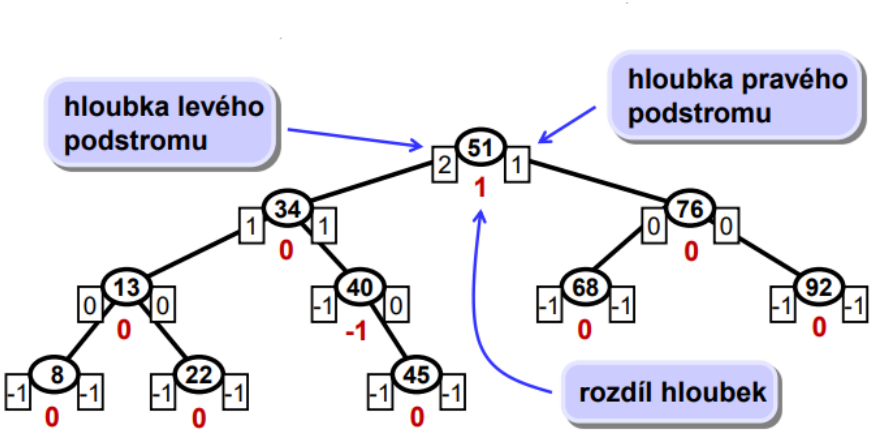


	BVS s n uzly	
Operace	**Obecný**	**Vyvážený**
Find	$\Theta(n)$	$\Theta(\log n)$
Insert	$\Theta(n)$	$\Theta(\log n)$
Delete	$\Theta(n)$	$\Theta(\log n)$

AVL strom

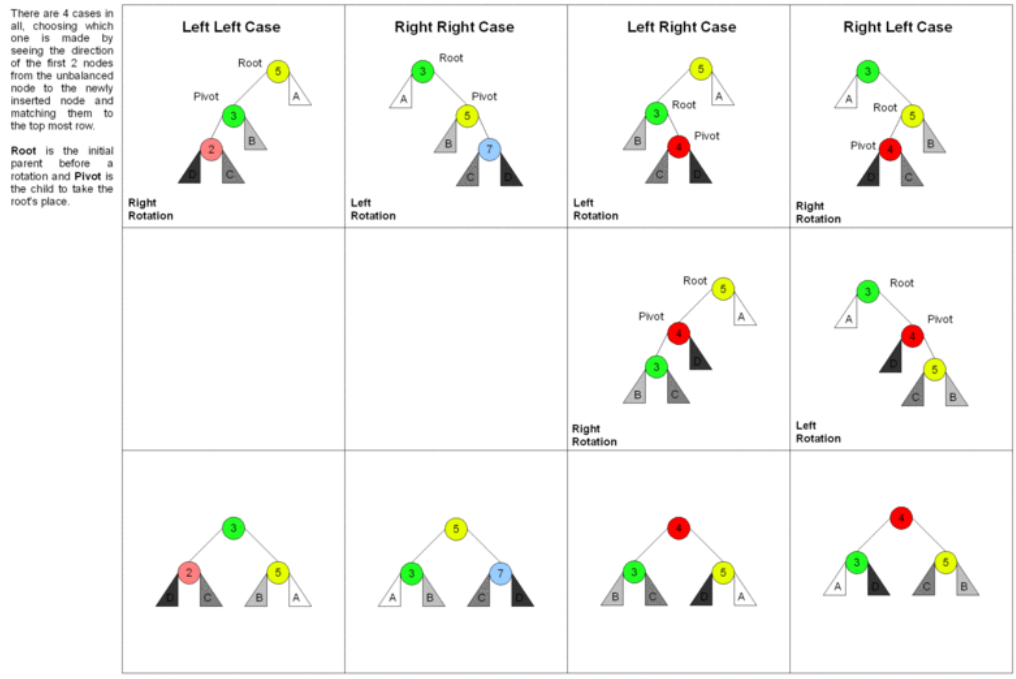
AVL strom je datová struktura pro uchovávání údajů a jejich vyhledávání. Pracuje v logaritmicky omezeném čase. Jedná se o **samovyvažující se binární vyhledávací strom**.

- Vrchol má maximálně dva následníky (plyne z vlastností binárního stromu, kterým AVL strom je).
- V levém podstromu vrcholu jsou pouze vrcholy s menší hodnotou klíče
- V pravém podstromu vrcholu jsou pouze vrcholy s větší hodnotou klíče
- **Délka nejdelší větve levého a pravého podstromu se liší nejvýše o 1 (vyvážení AVL stromu).**



Poslední vlastnost je demonstrována na následující dvojici stromů.

1. **Vyhledávání**
   Totožné jako u vyhledávání v BVS.
2. **Přidání uzlu**
   Totožné jako u přidávání v BVS. Strom se ale po přidání může stát nevyváženým, proto je potřeba provést vyvažovací rotace.
3. **Rotace**

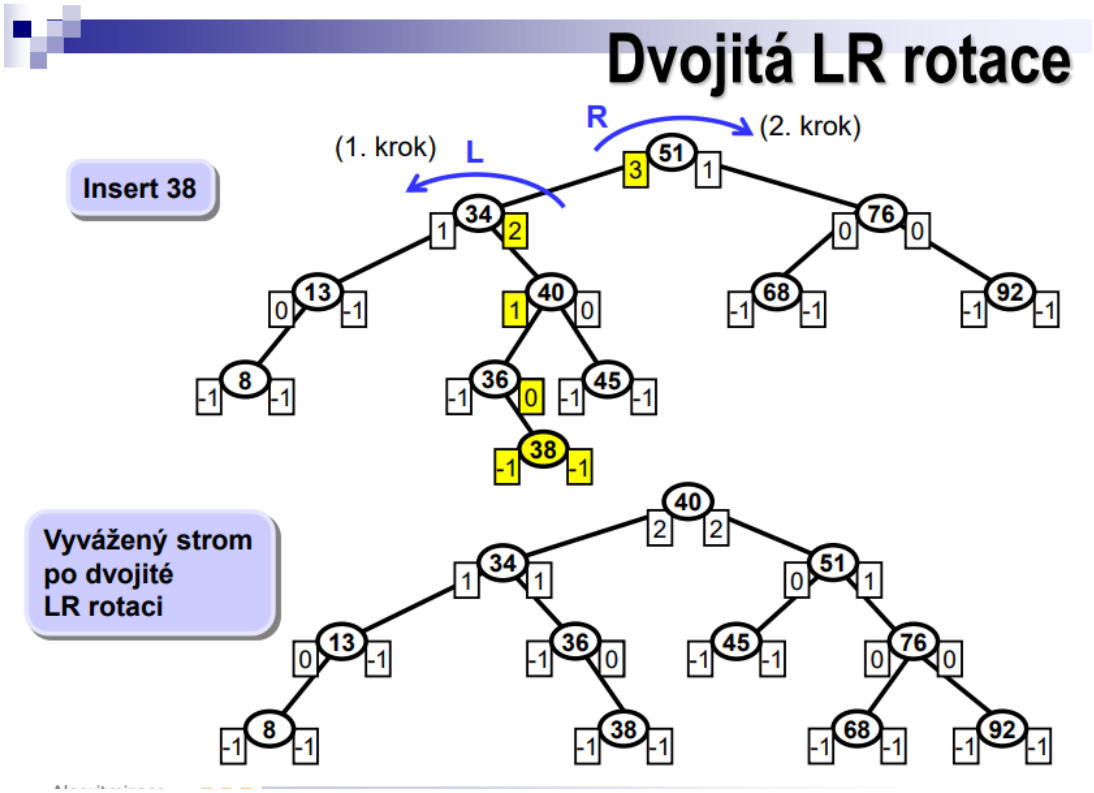


4. Ve směru rotace nahradíte špatně vyvážený uzel potomkem.
5. Pokud by se stalo, že by nově dosazený vrchol měl 3 potomky, tak jeho původního potomka umístíte pod právě sesazený vrchol.

Kdy jakou použít? Od přidaného (nebo smazaného, viz dále) uzlu postupujeme směrem ke kořenu a aktualizujeme hloubky podstromů v každém navštíveném uzlu.

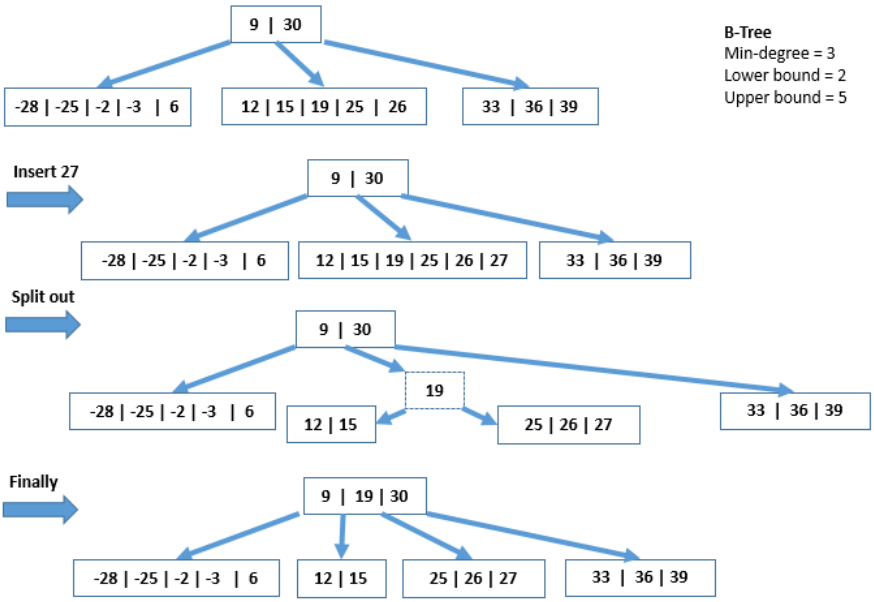
- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli dvěma hranami doleva nahoru, provedeme v tomto uzlu L rotaci.
- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli dvěma hranami doprava nahoru, provedeme v tomto uzlu R rotaci.
- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli hranami doleva a pak doprava nahoru, provedeme v tomto uzlu LR rotaci (tedy nejprve levou a pak pravou).
- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli hranami doprava a pak doleva nahoru, provedeme v tomto uzlu RL rotaci.

Grafický příklad LR rotace



4. **Odstranění uzlu**
   Opět totožné jako u odstranění v BVS. Poté postupujeme od místa smazání nahoru ke kořeni a aktualizujeme výšky podstromů v každém uzlu. V případě nevyváženosti použijeme stejná pravidla jako u vkládání prvku.

B-strom

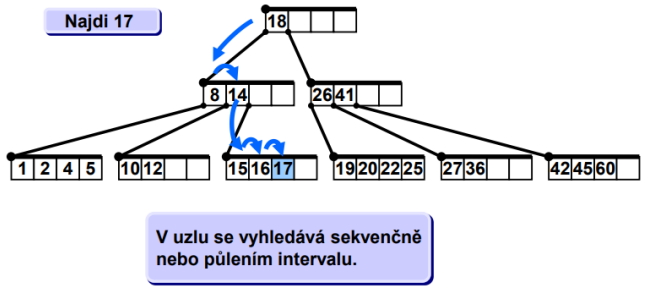


B-strom je druh stromu, který ale **nesouvisí** s binárními stromy. Jeho definice

- Kořen má nejméně dva potomky, pokud není listem
- Každý uzel kromě kořene a listu má nejméně $\lceil \frac{n}{2} \rceil$ a nejvýše n potomků.
- Každý uzel kromě kořene má nejméně $\lceil \frac{n}{2} \rceil - 1$ a nejvýše $n - 1$ položek.
- Všechny cesty od kořene k listům jsou stejně dlouhé

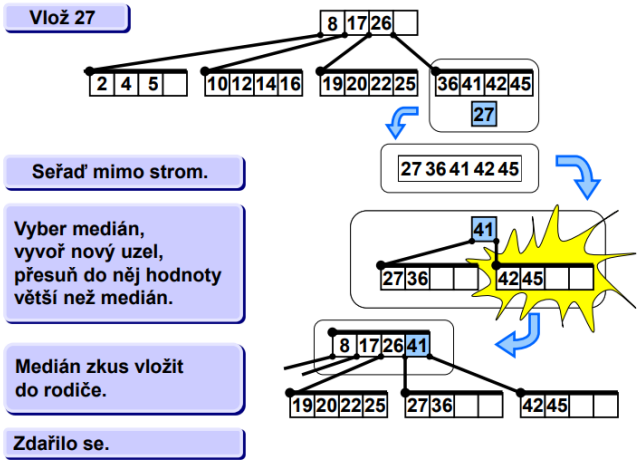
B-strom je díky těmto vlastnostem vyvážený, operace přidání, vyjmutí i vyhledávání tedy probíhají v logaritmickém čase.

Operace Find

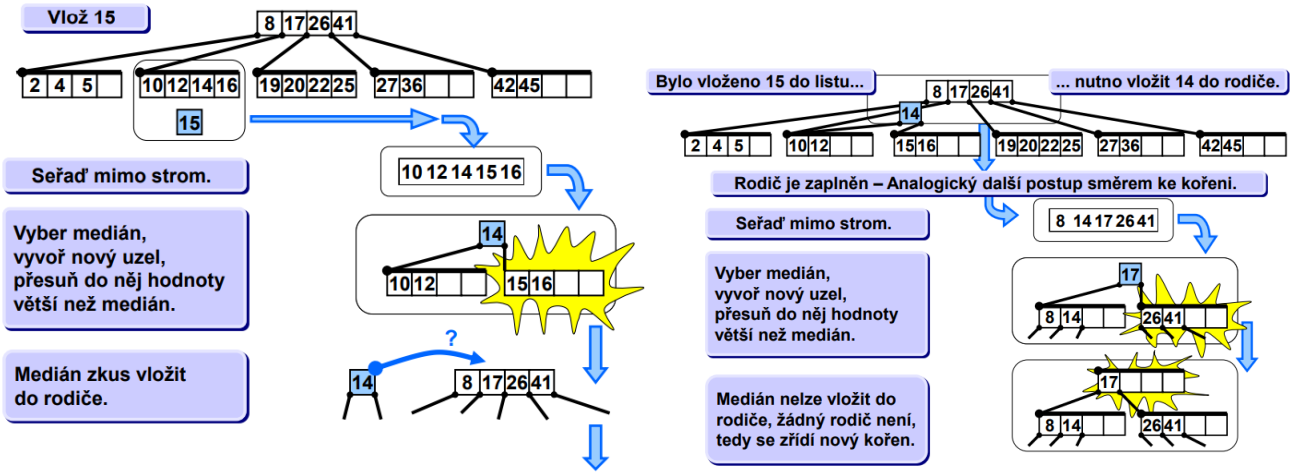


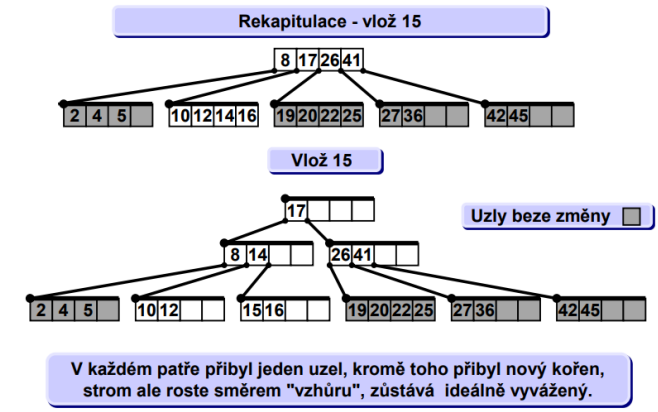
Operace Insert

- Pokud vložením nepřekročíme kapacitu uzlu, tak neprobíhá žádné vyvážení.
- Pokud překročíme tak:



- Pokud by se to stalo vícenásobně tak:





Srovnání stromů

	AVL s n uzly	BVS s n uzly	B-strom s n uzly	
Operace	Vývážený	Možná nevyvážený	Vyvážený	Vyvážený
Find	O(log(n))	O(n)	O(log(n))	O(log(n))
Insert	Θ(log(n))	O(n)	Θ (log(n))	Θ(log(n))
Delete	Θ(log(n))	O(n)	Θ(log(n))	Θ(log(n))

Pár poznámek na závěr

- označení binární strom je nadmnožina pro binární vyhledávací strom a AVL strom
- Vyváženost zajišťuje AVL stromu lepší asymptotickou složitost. V tabulce srovnávající jednotlivé stromy je uveden sloupec „vyvážený binární vyhledávací strom", nicméně to, že náš BVS na vstupu bude skutečně vyvážený, nemáme zaručeno. U AVL stromu to zaručeno je.
- B-strom a AVL strom mají stejnou složitost, každý se ale hodí v jiné situaci. Využití b-stromu může být v aplikacích, kdy není celá struktura uložena v paměti RAM, ale v nějaké sekundární paměti, jako je pevný disk (například databáze). Protože přístup do tohoto typu paměti je náročný na čas (hlavně vyhledání náhodné položky), snažíme se minimalizovat počet přístupů do této paměti.

Algoritmy řazení: Insert Sort, Selection Sort, Bubble Sort, QuickSort Merge Sort, Halda, Heap Sort, Radix sort, Counting Sort.

V popiscích jednotlivých algoritmů se vyskytují následující hesla a charakteristiky:

Stabilní/nestabilní algoritmus
Stabilním algoritmus je ten, který zachovává pořadí stejných prvků (mají stejnou hodnotu), tak jak byly na vstupu v seznamu. Seřadí je na odpovídající místo, ale v rámci stejných prvků zůstanou ve stejném pořadí. U nestabilního algoritmu toto nezaručíme.
Nestabilní: Selection sort, quick sort, heap sort
Stabilní: Merge, Bubble, Insertion, Radix

Insert Sort

Prohlásím první prvek za seřazený. Vezmeme si další prvek a po dvojicích ho porovnáváme s předešlými "seřazenými" prvky. Pokud je předešlý prvek větší, tak je vyměníme a opakujeme dokud to platí. Jakmile toto neplatí, vezmeme první neseřazený prvek a postupujeme stejným způsobem.

Selection Sort

Pamatuji si ukazatel na první neseřazený prvek v seznamu. Procházím seznam dokud nenajdu nejmenší prvek a ten vyměním za první neseřazený prvek a posunu si ukazatel na začátek o jedna.

Bubble Sort

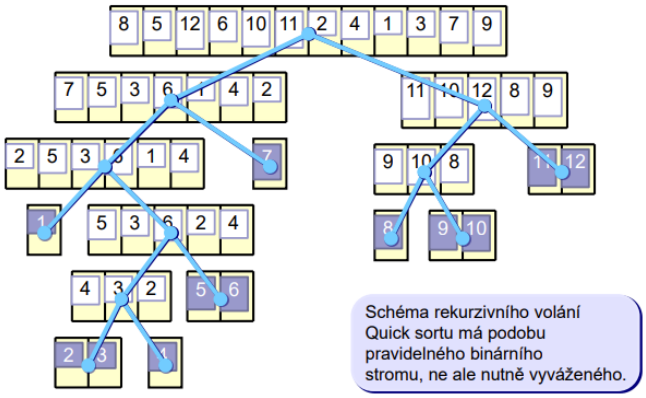
Několikrát se prochází celý seznam po dvojících. Tyto dvojice se prohodí pokud pokud je první prvek větší než druhý. Takto se prochází celý seznam dokud není seřazený.

Quicksort

Quicksort je velmi rychlý **nestabilní** řadící algoritmus na principu **rozděl a panuj** s asymptotickou složitostí $O(n^2)$ a s očekávanou složitostí $O(n \cdot log(n))$
.

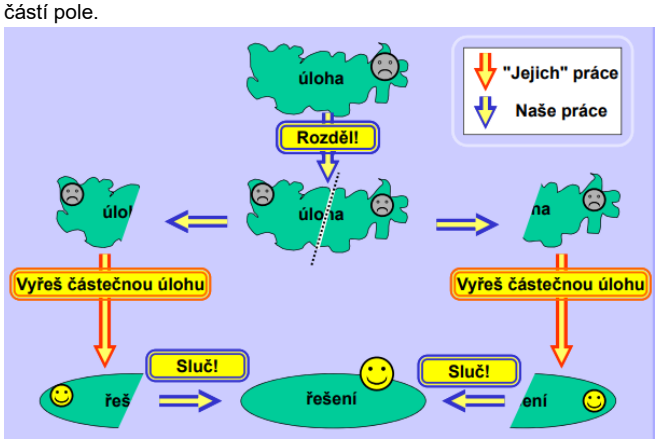
Průběh algoritmu

1. Levý index se nastaví na začátek zpracovávaného úseku pole, pravý na jeho konec, zvolí se pivot, nejjednodušeji tak, že se vybere prvek na začátku zpracovávaného úseku. Levý index se pohybuje doprava a zastaví se na prvku větším nebo rovném pivotovi. Pravý index se pohybuje doleva a zastaví se na prvku menším nebo rovném pivotovi.
2. Pokud je levý index ještě před pravým, příslušné prvky se prohodí, a oba indexy se posunou o 1 ve svém směru. Jinak pokud se indexy rovnají, jen se oba posunou o 1 ve svém směru.
3. Cyklus se opakuje, dokud se indexy neprekříží, tj. pravý se dostane pred levého.
4. Následuje rekurzivní volání (zpracování „malých" a „velkých" zvlášť) na úsek od začátku do pravého indexu včetně a na úsek od levého indexu včetně až do konce, má-li príslušný úsek délku větší než 1.



Mergesort

Mergesort je **stabilní** řadicí algoritmus typu **rozděl a panuj** s asymptotickou složitostí O(n*log(n)). Merge sort pracuje na bázi **slévání** již seřazených částí pole.

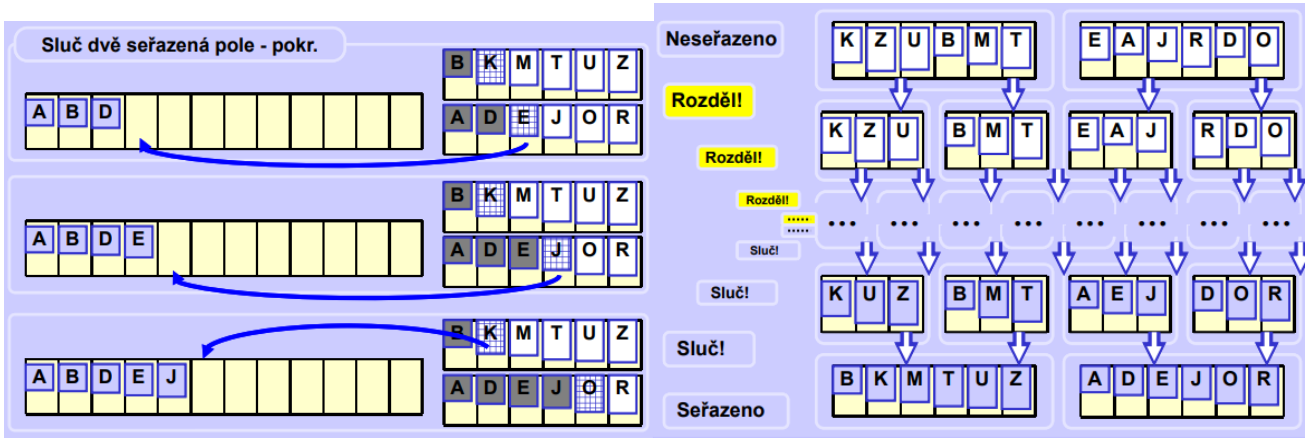


Merge

Proces merge nebo-li slévání probíhá následovně. Máme dva seznamy, u kterých víme, že jsou seřazeny ve stejném pořadí. Postupně je procházíme a díváme se, který z právě iterovaných členů je větší (případně menší, záleží, jak srovnávám). Ten větší (resp. menší) z dvojice vložíme do pomocného seznamu. Takto se dostaneme do fáze, kdy jeden ze seznamů, kterými iterujeme, je už prázdný. Můžeme tedy zbytek druhého vzít a celý ho zkopírovat na konec do pomocného seznamu.

Průběh algoritmu

Algoritmu dodáme array, který postupným půlením rekurzivně rozdělíme do nejmenších skupin, tedy dvojic a případně lichého členu. Pak se aplikuje proces merge na tyto malé jednotky, a ve stacku, který se nám během rekurze vytvořil, skočíme o úroveň výš, kde sléváme podseznamy délky 4, případně 3. Takto skáčeme až do chvíle, kdy se ocitneme v hlavní smyčce programu a sléváme levou a pravou polovinu seznamu, který nám byl vložen na vstupu.



Heapsort

Heapsort (řazení haldou) je jedním z nejefektivnějších řadících algoritmů založených na porovnávání prvků s asymptotickou složitostí $O(n \cdot log(n))$. Heapsort je **nestabilní**.

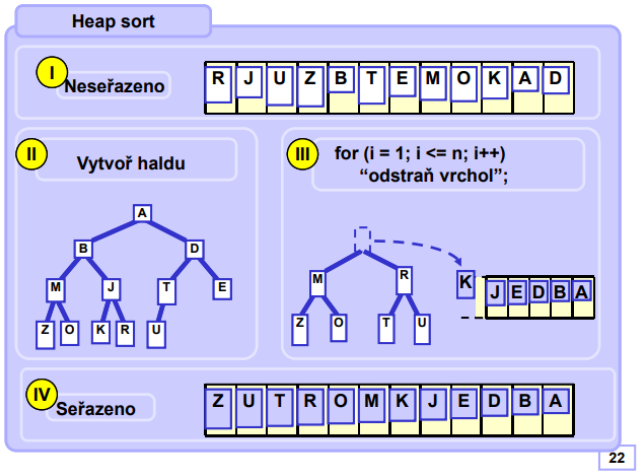
Vlastnosti haldy

Halda je binární strom s rekurzivní vlastností býti haldou. Rekurzivita vlastnosti značí, že i každý podstrom haldy je taktéž haldou. V každé haldě také platí, že otec má vždy vyšší (v některých definicích nižší) hodnotu než jeho potomci. Při její implementaci polem pak také platí, že pokud je otec na indexu i , tak jsou jeho potomci na indexech 2i+1 a 2i+2 (indexováno od 0). Z tohoto uspořádání plyne, že tvar haldy je pyramida s částečně useknutou pravou stranou základny.

Operace delete

Odebírá se vždy kořen haldy, v takovém případě se místo něj vloží poslední prvek v haldě. Tím se však poruší vlastnost haldy a tak se porovná nový kořen s jeho dětmi a prohodí se podle hodnoty. To rekurzivně pokračuje dokud není vlastnost haldy obnovena.

Průběh algoritmu

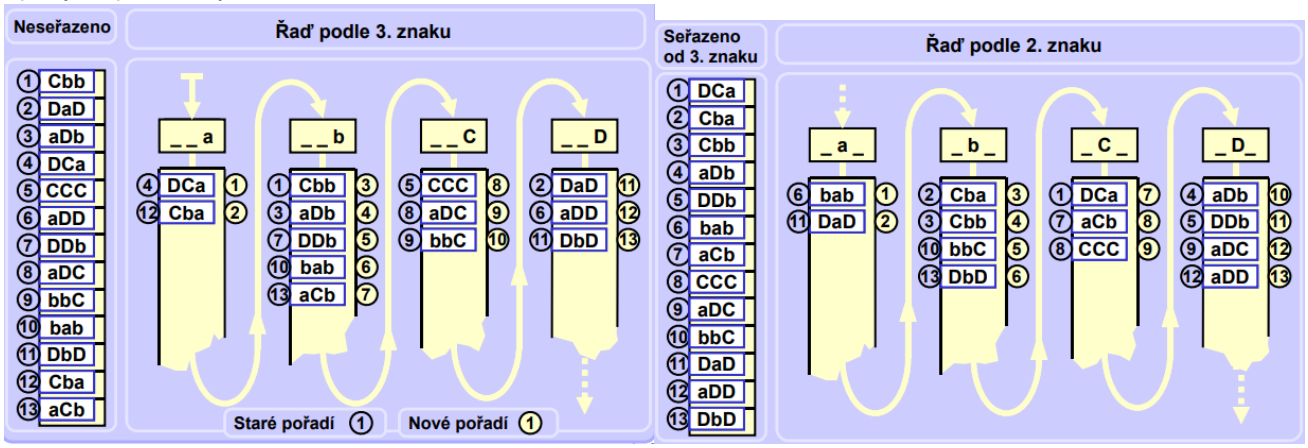


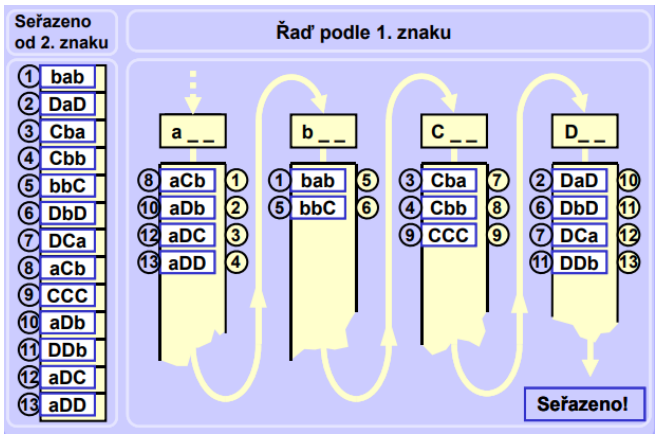
Radixsort

Princip radix sortu vychází přímo z definice stabilního řazení – řadicí algoritmus je stabilní, pokud zachovává pořadí klíčů, které mají stejnou hodnotu (tj. pokud struktury seřadíme napřed dle klíče **A**, poté podle klíče **B**, tak jsou seřazeny podle **B**, a kde jsou si hodnoty **B** rovny, tam jsou struktury v pořadí daném klíčem **A**).

Postup

1. Začneme s poslední číslicí a pole seřadíme podle ní (například pomocí Counting Sort nebo separátních polí pro každou číslici)
2. Opakujeme pro všechny číslice





Implementuje se pomocí pomocných polí s indexy pořadí v daných polí podle znaku.
(https://cw.fel.cvut.cz/b211/_media/courses/b4b33alg/alg09_2011.pdf)

Counting sort

Vytvoříme si pole četnosti každého čísla v seznamu (zjistíme max a min). Pak jednou projedeme polem a podle výskytu čísla upravíme četnost. Ve finále projdeme pole četností a do výstupního pole dáme tolik aktuálního prvku jako je jeho četnost.

Srovnání složitostí

Orientační přehled vlastností řadících algoritmů

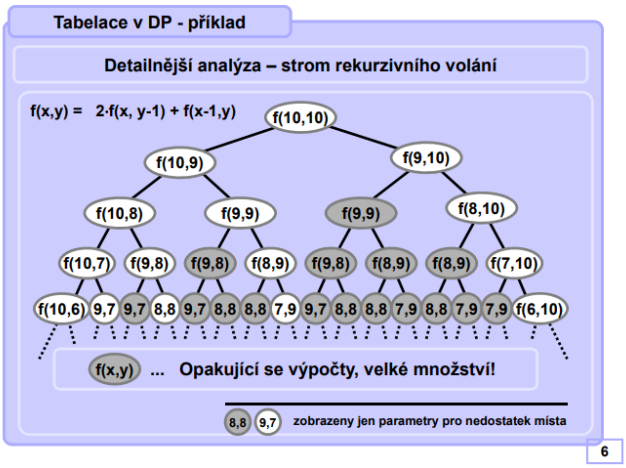
Velikost pole n	Nejhorší případ	Nejlepší případ	"typický", "běžný" případ	Stabilní	Všeobecné
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Ne	Ano
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Ano	Ano
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Ano	Ano
Quick sort	$\Theta(n^2)$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ne	Ano
Merge sort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ano	Ano
Heap sort	$\Theta(n \cdot \log(n))$	$O(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ne	Ano
Radix sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano	Ne
Counting sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano	Ne

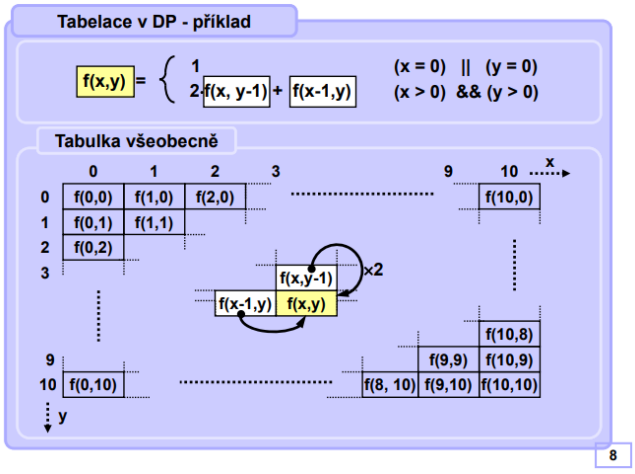
Dynamické programování, struktura optimálního řešení, odstranění rekurze.

Je to všeobecná strategie pro řešení optimalizačních úloh.
Významné vlastnosti:

- Hledané optimální řešení lze sestavit z vhodně volených optimálních řešení téže úlohy nad redukovanými daty.
- V rekurzivně formulovaném postupu řešení se opakovaně objevují stejné menší podproblémy. DP umožňuje obejít opakovaný výpočet většinou jednoduchou tabelací výsledků menších podproblémů, tedy volně řečeno, přepočítáním menších výsledků.

V následujícím příkladu je ukázána složitost rekurzivního volání, oproti přepočítání tabulky, kde vycházíme z předešle vypočítaných dat.





Nejdelší společná podposloupnost

Mějme dvě posloupnosti písmen a chceme najít nejdelší společnou podposloupnost.
Například u posloupností {BDCABA} {ABCBDAB} je řešení {BCBA}.
Časová složitost O(mn), kde m a n jsou délky řetězců.

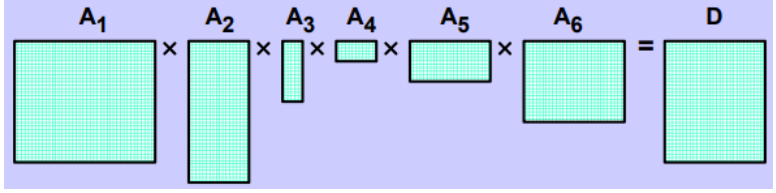
Nalezení výpočtu hodnoty optimálního řešení zdola nahoru.

```
Function DÉLKA-NSP(X, Y)
1)   m ← length[X];
2)   n ← length[Y];
3)   for i ← 1 to m do
4)        c[i, 0] ← 0;
5)   for j ← 0 to n do
6)        c[0, j] ← 0;
7)   for i ← 1 to m do
8)        for j ← 1 to n  do
9)             if x_i = y_j
10)            then  { c[i, j] ← c[i - 1, j - 1] + 1;
11)                    b[i, j] ← "↖"; }
12)            else if c[i - 1, j] ≥ c[i, j - 1];
13)            then { c[i, j] ← c[i - 1, j];
14)                   b[i, j] ← "↑"; }
15)            else { c[i, j] ← c[i, j - 1];
16)                   b[i, j] ← "←"; }
17)  return c and b;
```

		j	0	1	2	3	4	5	6
i			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

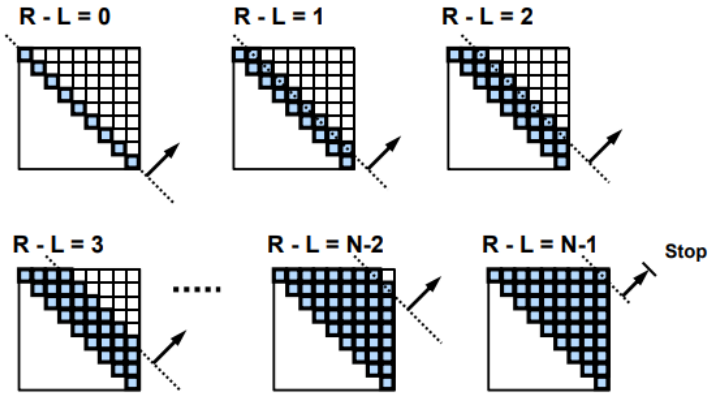
Optimální násobení matic

Různé pořadí násobení matic vyprodukuje různé množství operací a proto je vhodné počet operací minimalizovat.

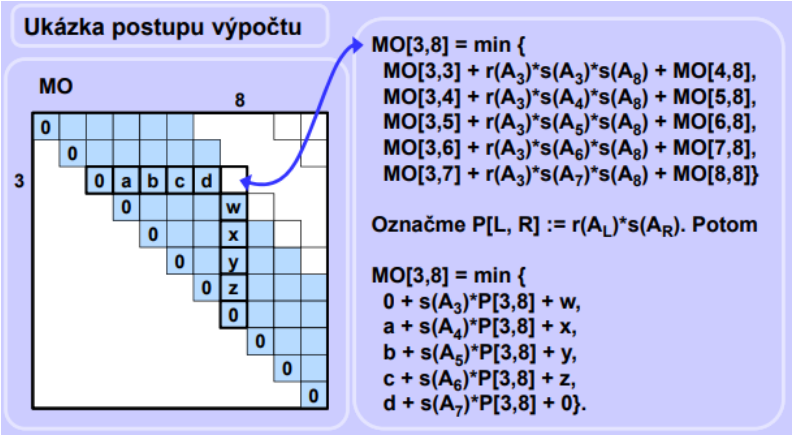


metoda	Výraz	Počet operací
zleva doprava	$((((A_1 \times A_2) \times A_3) \times A_4) \times A_5) \times A_6$	43 500
zprava doleva	$A_1 \times (A_2 \times (A_3 \times (A_4 \times (A_5 \times A_6))))$	47 500
nejhorší	$A_1 \times ((A_2 \times ((A_3 \times A_4) \times A_5)) \times A_6)$	58 000
nejlepší	$(A_1 \times (A_2 \times A_3)) \times ((A_4 \times A_5) \times A_6)$	15 125

Algoritmus probíhá pomocí sestrojení matice, kde na hlavní diagonále budou samé 0 (reprezentující počet operací na dosažení dané matice). Matice se pak naplňuje po diagonálách doprava nahoru

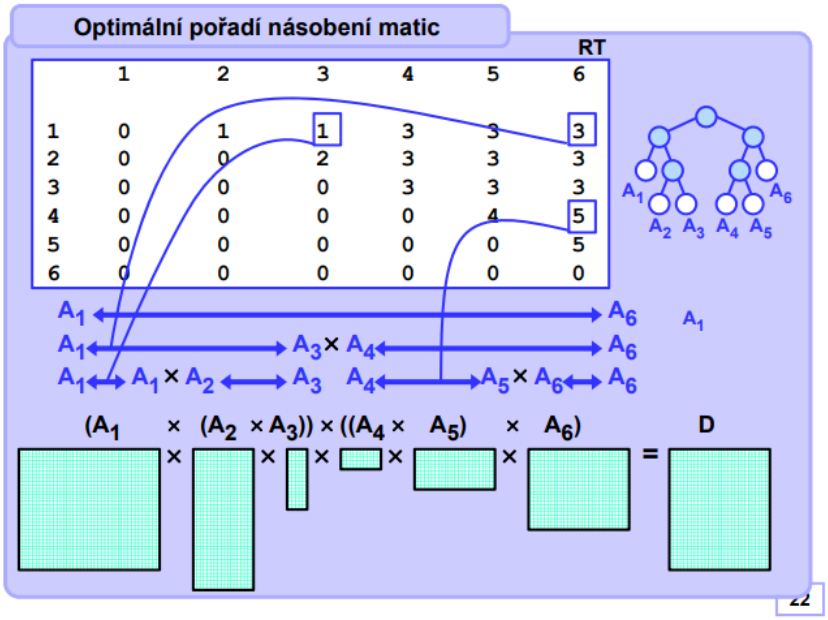


Pro každou buňku pak bereme minimum všech možných operací.



Nejmenší počet operací pak bude v pravém horním rohu.

Pokud chceme získat optimální uzávorkování, pamatujeme si v každé buňce i index matice (číslo sloupce), které bylo optimální a následujícím způsobem postupujeme:



Problém batohu.

Máme N předmětů a každý má nějakou váhu V_i a nějakou cenu C_i s tím, že máme batoh kapacity K. Naším úkolem je naplnit batoh tak, abychom maximalizovali cenu a nepřekročili kapacitu.

Rozdělujeme na:

- neomezenou variantu - lze použít každý předmět neomezeně
- 0/1 variantu - lze použít každý předmět nejvýše jednou

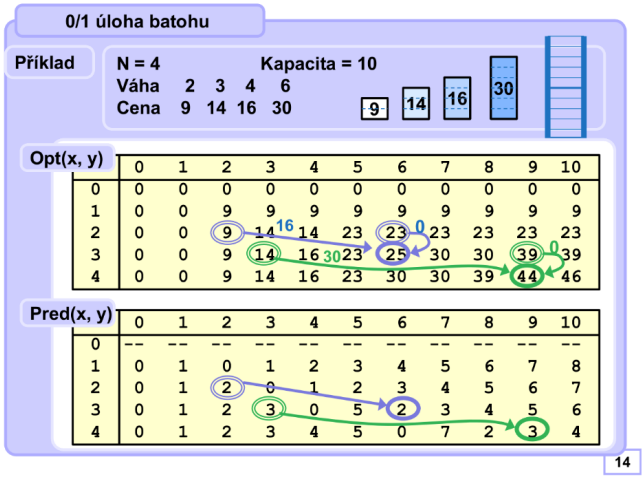
Neomezená varianta

Úlohu převedeme na DAG, kde vrcholy jsou kapacity batohu od 1 do K a hrany reprezentují přidání předmětu (cena hrany = cena předmětu).

0/1 varianta

Uděláme si tabulku, kde sloupce budou reprezentovat jednotlivé kapacity od 0 do K a řádky použití jednotlivých objektů.

Tabulku vyplňujeme následujícím způsobem: (https://bbb04.felk.cvut.cz/playback/presentation/2.3/9e304a8d9f75b500527e81f95fcf96123c8a1602-1607075941901?meetingId=9e304a8d9f75b500527e81f95fcf96123c8a1602-1607075941901, slide 114)



Rozptylovací tabulky (Hashing)

Základ datové struktury slovníku, kde operace search a insert mají konstantní složitost.

Hashovací funkce, řešení kolizí

Vypočítává adresu z hodnoty klíče, závislá na vlastnostech klíčů a jejich reprezentaci v paměti. Ideálně by měl výpočet být co nejjednodušší, náhodný, rovnoměrně by měl využívat adresní prostor a generuje minimum kolizí.

Řešení kolizí

Kolize nastávají, protože zobrazení z klíčů do adres hashovací funkce není prosté.

Zřetězené tabulky
Každá adresa si zachovává spojový seznam všech hodnot, které se namapují na danou adresu. Takže při kolizi se prvek zařadí na konec spojového seznamu. Insert je $O(1)$, search a delete v nejhorším případě $O(n)$.

Otevřené tabulky
Je dobré znát počet prvků předem (odhad). Tabulka může být více zaplněná než začne klesat výkonnost.
Pokud nastane kolize tak:

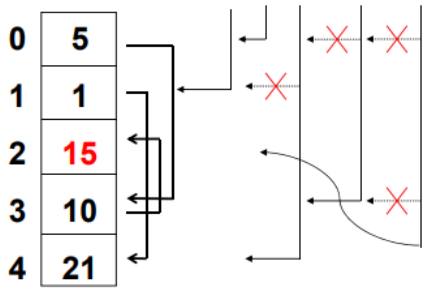
- Linear probing - prvek se buď vloží na pozici + "n", kde n je prvočíslo nebo číslo nesoudělné s velikostí pole, problém shlukování
- Double hashing - verze linear probing, kde se používá jedna hashovací funkce na určení adresy a druhá na určení "n"

Srůstající tabulky (coalesced hashing)

Standartní

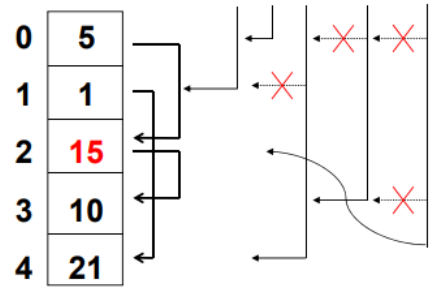
LISCH (late insert standard coalesced hashing)

- Kolizní prvky se lineárně přidávají na konec tabulky
- Každý prvek, se kterým je kolize má v sobě ukazatel na místo, kde se kolidující prvek nachází
- Pokud nám nastane, že máme řetěz ukazatelů, tak nový prvek přidáme na konec
  - $h(k) = k \bmod 5$
  - posloupnost: 1, 5, 21, 10, 15



EISCH (early insert standard coalesced hashing)

- Funguje stejně jako LISCH, ale pokud nám nastane, že máme řetěz ukazatelů, tak nový prvek přidáme na začátek řetězu
  - $h(k) = k \bmod 5$
  - posloupnost: 1, 5, 21, 10, 15



S pomocnou pamětí

LICH a EICH

- Analogické k LISCH a EISCH akorát máme pomocnou paměť sklepa na konci určenou pouze pro kolizní prvky

VICH

- připojuje prvek na konec řetězce, pokud řetězec končí ve sklepě (EICH), jinak na místo, kde řetězec opustil sklep (LICH)

Univerzální hashování

Každá hashovací funkce má slabá místa, kdy pro různé klíče dává stejnou adresu. Proto je výhodné přizpůsobit hashovací funkci právě zpracovávaným klíčům.

Místo jedné hashovací funkce máme konečnou množinu funkcí mapujících U do intervalu $\{0, 1, \ldots, m-1\}$. Tato množina je univerzální, pokud pro různé klíče $x, y \in U$ vrací stejnou adresu $h(x) = h(y)$ přesně v $|H|/m$ případech. Pravděpodobnost kolize při náhodném výběru funkce $h(k)$ je tedy přesně $1/m$.

Při spuštění programu jednu náhodně zvolíme. Funkci pak náhodně měníme jen v případě, že počet kolizí převyšuje přípustnou mez. V tomto případě je samozřejmě potřeba přehashovat celou tabulku.