Standardní grafové úlohy s polynomiální složitostí řešení. Kombinatorické a číselně teoretické algoritmy, izomorfizmus, prvočíselnost. Vyhledávací stromy a jejich využití. Vyhledávání v textu založené na konečných automatech. [A4M33PAL](https://www.fel.cvut.cz/cz/education/bk_peo/predmety/12/58/p12581604.html)

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html> vizualizace algoritmů

[Asymptotická složitost](#_kdsyg19bozuw)

[1.1 Amortizovaná složitost](#_yy0ls863p810)

[2 Polynomiální algoritmy](#_qmnj7222tgl4)

[2.1 Algoritmy pro hledání minimální kostry](#_yu65mhvk0zip)

[2.2 Algoritmy pro hledání silně souvislých komponent](#_u09xsn27bpxv)

[2.3 Další grafové algoritmy](#_mcgz97wtjjwl)

[2.4 Haldy](#_3xcicmf93dl3)

[3. Kombinatorické a číselně teoretické algoritmy, izomorfizmus, prvočíselnost](#_ebht8330m4ji)

[3.1 Generování a enumerace datových struktur a kombinatorických objektů](#_89dmit4y6cnk)

[3.2 Generátory náhodných čísel](#_9m3f00ibtm70)

[3.3 Prvočísla](#_q9o5klfiglbq)

[3.3.1 Pojmy](#_hemh81ypun85)

[3.3.2 Algoritmy pro prvočísla](#_edr05s8b3g0d)

[3.3.3 Algoritmy pro faktorizaci celých čísel](#_2x8hc3311n72)

[3.4 Izomorfismus](#_5dwiz5wthjfy)

[4. Vyhledávací stromy a jejich využití](#_sghsuro7mdut)

[4.1 BVS (BST) - Binary search tree](#_dvr6k8a89o6o)

[4.2 AVL](#_zg6ay2t1pcpv)

[4.3 Skip list](#_vqbqg9lxcrxh)

[4.4 B a B+](#_1nfm8bosk9vh)

[4.4.1 B tree](#_52wmxulgkzg2)

[4.4.2 B+ tree](#_f8j3azw2zwgt)

[4.6 R-B tree](#_onzed7yqj2w8)

[4.7 Splay tree](#_p95qij54lplj)

[4.8 2-3-4 strom](#_6ow29bt9q1l6)

[4.9 Porovnání stromů](#_56lsixtojgrj)

[4.8 K-D strom - hledání ve více dimenzích](#_pserfqbcn8du)

[Operace](#_7vni2e7x6izg)

[4.4 Binary trie](#_6o5ch29nn8mi)

[4.5 Patricia trie](#_40h8jszesdd2)

# 

# Asymptotická složitost

𝑓(𝑛) ∈ Ο(𝑔(𝑛))

* ,,funkce f je omezená funkci g shora” = f roste asymptoticky pomaleji než g
* (∃𝑐 > 0) (∃𝑛0) (∀𝑛 > 𝑛0) ∶ |𝑓(𝑛)| ≤ |𝑐 ∙ 𝑔(𝑛)|

𝑓(𝑛) ∈ Ω(𝑔(𝑛))

* ,,funkce f je omezená funkcí g zdola” = f roste asymptoticky rycheleji než g
* (∃𝑐 > 0) (∃𝑛0) (∀𝑛 > 𝑛0) ∶ |𝑓(𝑛)| ≥ |𝑐 ∙ 𝑔(𝑛)|

𝑓(𝑛) ∈ Θ(𝑔(𝑛))

* ,,funkce f je omezená funkcí g shora i zdola” = f roste asymptoticky stejně rychle jako funkce g
* (∃𝑐1, 𝑐2 > 0) (∃𝑛0) (∀𝑛 > 𝑛0) : |𝑐1 ∙ 𝑔(𝑛)| <= 𝑓(𝑛) <= |𝑐2 ∙ 𝑔(𝑛)|
* pozor tady na ty dvě různá céčka

*Malou omegu a omikron jsme v PALech neřešili*

*Příklad: Dvourozměrné pole čísel - asymptotická složitost vyhledání čísla?*

* *vyšší je například O((M+N)^2) nebo O(max(M,N)^2)*
* *nižší je například* Ω*(1) nebo* Ω(M)
* hledáme však vždy **nejpřesnější možné vyjádření (omezení)** v tomto případě Θ(M\*N)

## 1.1 Amortizovaná složitost

* čas potřebný pro provedení sekvence operací je zprůměrován přes všechny provedené operace
* jedna operace může být drahá, ale když operací provedeme více a čas zprůměrujeme, může být čas relativně malý
* garantuje průměrnou výkonnost (performance) operace v nejhorším případě

*Příklad: Typický případ s Array Listem*

* *insert do listu stojí O(1), ale když přesáhneme jeho délku musí se array list zvětšit (zdvojnásobit) a elementy překopírovat, což stojí nejhůře O(N). Asymptotická složitost by tedy byla O(N)*
* *amortizovanou vypočteme pozpátku - poslední insert, který vedl k realokaci stál nejvýše 2n + n + n = O(N). 2n kvůli alokaci pole, které je 2x vetší než původní, n kvůli kopírování n prvků do nového pole a druhé n kvůli dealokaci starého pole. Předtím se n/2 provedl jen insert, který stál konstantní čas. Poté (n/2 + 1)-ní Insert zase musel realokovat atd. atd. ve výsledku se při volání n inser operací provedlo nejvýše*
* *cn + cn/2 +cn/4 + … + 1 operací + O(n), kde cn, cn/2 atd. jsou časy pro realokaci a O(n) je čas vložení n prvků. Dohromady je to asymptoticky O(n), což když zprůměrujeme přes n operací, je amortizovaná složitost O(1).*

# 2 Polynomiální algoritmy

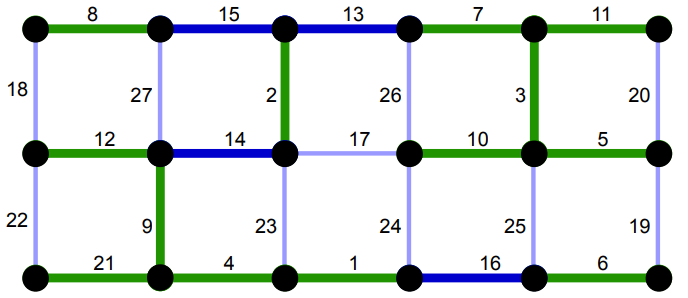
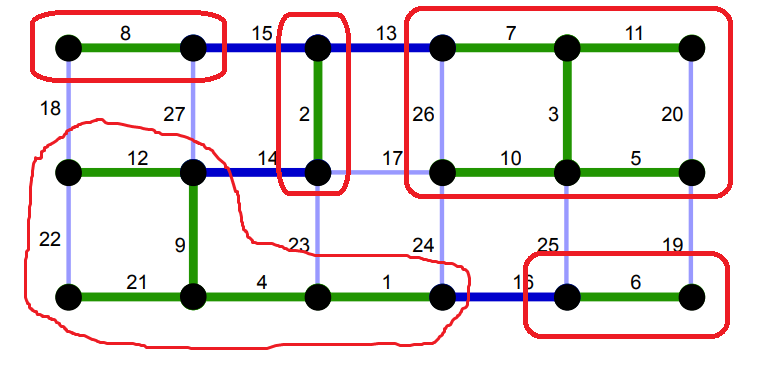
Řez grafu (cut of graph) - podmnožina hran taková, že k nim existuje podmnožina vrcholů, ze kterých vede hrana z této podmnožiny do vrcholu, který v podmnožině není. Hrany v podstatě rozdělují graf na dva grafy.

### 2.1 Algoritmy pro hledání minimální kostry

**Jarníkův (Primův) algoritmus (**[**YouTube**](https://www.youtube.com/watch?v=cplfcGZmX7I)**)**

* slouží nalezení minimální kostry grafu
* Postup:
  + začni v libovolném vrcholu a přidej ho do kostry
  + opakuj dokud nejsou navštíveny všechny vrcholy (resp. dokud nebude mít kostra stejně vrcholů jako graf):
    - vyber nejlevnější hranu, která vede **z kteréhokoli vrcholu, které už jsou v kostře do vrcholu, který ještě není v kostře** a přidej vrchol a hranu do kostry
* V každé iteraci je přidaný do kostry jeden vrchol = algoritmus musí skončit zákonitě po |V| krocích
* Kostra je 100% strom, neboť díky checku, že koncový vrchol nejlevnější hrany ještě není v kostře nemůže vzniknout cyklus
* Složitost: záleží na reprezentaci grafu -
  + matice sousednosti O(|V|2)
  + binární halda + adjacency list - O((V + E) log(V)) = O(E\*logV)
  + fibonacciho halda + adjacency list - O(E + Vlog(V))

**Borůvkův algoritmus (**[**YouTube**](https://youtu.be/czcf73b0Ga0?t=2m11s)**)**

* slouží nalezení minimální kostry grafu
* lze použít jen, když jsou všechny ceny **rozdílné**
* pracuje s komponentami grafu
* Postup:
  + Na začátku vložíme do kostry všechny vrcholy grafu, ale žádné hrany
  + Opakujeme, dokud v kostře existují alespoň dvě rozdílné souvislé komponenty (tj. dokud není graf celý ,,propojený”)
    - pro všechny komponenty grafu vybereme hranu takovou, která komponentu spojí s jinou komponentou a je nejlevnější
    - přidej hrany do kostry

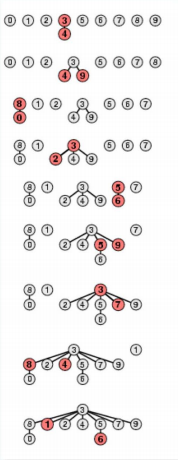
V prvním kroce hledáme nejlevnější hranu z každého vrcholu - reprezentováno zeleně. To nám pak vytvoří komponenty, viz. druhý obrázek. Další vybraná vyhraná množina hran je označena modře. Potom už je hotovo, neboť máme už jen jednu komponentu.

* Složitost: **O(|E|\*log|V|)** 
  + algoritmus beží v maximálně log|V| iteracích, neboť na začátku máme tolik komponent kolik je vrcholů a v nejhorším případě se po doběhnutí jedné iterace vrcholy propojí alespoň s jedním dalším vrcholem = počet komponent je poloviční
  + každá iterace algoritmu stojí O(|E|), neboť pro výběr správných hran, musíme projet hrany všechny (u nich zjišťujeme, jestli koncové vrcholy jsou ve stejné komponentě nebo ne a pokud ne, tak jestli je ta hrana nejlevnější)

**Kruskalův algoritmus (**[**YouTube**](https://www.youtube.com/watch?v=71UQH7Pr9kU)**)**

* slouží nalezení minimální kostry grafu
* Postup:
  + Seřaď hrany podle cen od nejnižší po nejvyšší
  + Do kostry dej vrcholy grafu bez hran
  + Projedeme (popořadě) seřazené hrany
    - hranu přidáme do kostry pokud nevytvoří cyklus (neboli kostra zůstane acyklickým grafem)
    - neboli pokud nejsou vrcholy ve stejné komponentě, tak hranu můžeme přidat, jinak ne (viz union find)
    - Složitost - O(|E|\*log|E|)
      * sesortování hran stojí O(|E|\*log|E|)
      * |E| iterací (pro každou hranu jedna iterace)
      * v každé iteraci je nutné udělat union-find (složitost O(log|V|)

**Union-Find**

* řeší zda dva vrcholy patří do stejné komponenty souvislosti
* **FIND** vrací reprezentanta souvislé komponenty
* **UNION** sloučí dvě souvislé komponenty
* Implemetace pomocí pole reprezentantů
  + každý vrchol má zapsán index (číslo) vrcholu, který je reprezentantem souvislé komponenty, ve které se nachází
  + find jenom pak čte z pole
  + k unionu vrcholů u a v se udělají dvě find operace a pokud vrátí rozdílné reprezentanty, tak se reprezentant **u** přepíše na reprezentanta **v** 
* Implementace orientovaným stromem
  + každá komponenta je reprezentovaná jako strom, kde hrany vedou směrem nahoru ke kořeni - kořen je reprezentant
  + FIND vyleze nahoru ke kořeni a vrátí jej jako reprezentanta
  + UNION - zase dva findy a pokud jsou rozdílné naváže se menší komponenta na tu větší (kořen bude dítětem kořene druhé komponenty)
* Složitost: O(log|V|)

### 2.2 Algoritmy pro hledání silně souvislých komponent

**Kosaraju-Sharir algoritmus (**[**YouTube**](https://www.youtube.com/watch?v=RpgcYiky7uw)**)**

* slouží k nalezení množiny silně souvislých komponent
* Postup:
  + začneme v libovolném vrcholu
  + jdeme DFSkem do hloubky po hranách a vrcholy označujeme jako OPEN, jdeme tak dlouho dokud nás hrana nedovede do již OPEN vrcholu. Až se to stane vrchol, ze kterého chceme vyjít označíme jako CLOSED, vrchol umístíme na zásobník a zkusíme jinou cestu do některého z nenavštívených vrcholů. Pokud už žádná není vracíme se z rekurze a obdobným stylem uzavíráme otevřené vrcholy, přičemž při každém uzavření vrcholu, umístíme vrchol na zásobník.
  + Poté co máme uzavřeny všechny vrcholy uzavřeny a na zásobníku, **obrátíme hrany v grafu** a začneme ze zásobníku odebírat
  + Pro každý vrchol ze zásobníku, provádíme stejný algoritmus jako výše, akorát již nic neumisťujeme na zásobník.
  + Poté co pro daný vrchol ze zásobníku dokončíme DFS a máme uzavřeny všechny nody, do kterých jsme se mohli dostat, získáváme silně souvislou komponentu
  + Popneme další vrchol, ze zásobníku a pokud ještě nebyl navštívený (tzn. není uzavřeny = nenachází se ještě v komponentě souvislosti), provedeme opět algoritmus výše
  + To děláme tak dlouho, dokud máme něco na zásobníku.

**Nejlépe je postup vidět na slidech -** [**z PALu**](https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/2012pal03.pdf)

* Složitost
  + adjacency list (Θ(|V| + |E|))
  + adjacency matrix (O(|V|2)
  + je tomu tak neboť tento algoritmus provádí 2x průchod celým grafem

**Tarjanův algoritmus**

* hledá komponenty silné souvislosti
* u každého vrcholu si ukládáme následující:
  + index vrcholu - unikátní ID vrcholu
  + lowlink - díky tomu budeme vědět do jaké SCC patří
  + pred - pointer na svého předka
  + instack - jestli je vrchol ve stacku
* Postup:
  + Začneme v libovolném vrcholu, označíme jej jako 1 a lowlink mu dáme také 1, vložíme ho do stacku. Jeho předek (pred bude null)
  + Poté procházíme DFSkem, vrcholy označíme o 1 vyšším číslem, nastavíme stejný lowlink, hodíme vrchol na stack a ukazatel na předka bude ukazovat na vrchol, odkud jsme přišli.
  + Až se nám stane, že bychom chtěli jít na něco, co **už je na stacku**, porovnáme lowlinky. Pokud je lowlink tohoto vrcholu, do kterého chceme jít menší, než lowlink vrcholu, ve kterém jsme, snížíme lowlink aktuálního vrcholu na tu menší hodnotu
  + Pokud už není kam jít vracíme se v rekurzi a proces opakujeme.
  + **Pokud už není kam jít a číslo vrcholu je stejné jako lowlink vypopujeme ze stacku vrcholy až do totoho vrcholu - nalezli jsme komponentu silné souvislosti**

**Postup lze zase dobře vidět na slidech -** [**z PALu**](https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/2012pal03.pdf)

* Složitost - graf se projde celý jen jednou
  + adjacency list (Θ(|V| + |E|))
  + adjacency matrix (O(|V|2)
  + složitost je asymptoticky stejná jako Kosaraju Sharir, ale díky průchodu grafu pouze 1x je tento algoritmus reálně rychlejší

### 2.3 Další grafové algoritmy

**Hledání Eulerova tahu (**[**Teorie YouTube**](https://www.youtube.com/watch?v=ycRuO-u6rt8)**)**

* graf má eulerův tah (tah = každá hrana se opakuje pouze 1) pokud je souvislý a má buď 0 (tah končí tam kde začal) nebo 2 (tah začíná a končí v jiném vrcholu) vrcholů lichého stupně. Ostatní mají sudý stupeň.
* asociace - domeček jedním tahem
* Postup:
  + začneme v libovolném vrcholu
  + vydáme se libovolnou hranou a odmažeme ji za sebou
  + když se dostaneme do bodu, kde už nelze odmazat hranu, vrátíme se odkud jsme přišli a přidáme orientovanou hranu ve směru opačnému k vracení (tj. pokud se vracíme z u do v, tak bude přidána orientovaná hrana z v do u). Tato hrana je součástí tahu
  + takto se dostaneme nakonec až do počátečního bodu a tah by měl být určen
* Složitost:
  + jeden průchod grafem
  + adjacency list (Θ(|V| + |E|))
  + adjacency matrix (O(|V|2)

**Hamiltnovská cesta - NPC problém**

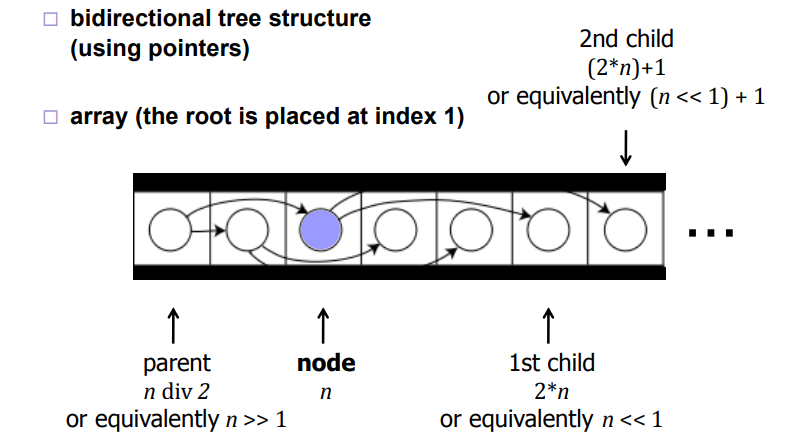
* **řeší zda graf obsahuje hamiltnovskou cestu = cesta, kdy projdeme všechny vrcholy grafu právě jednou**

### 2.4 Haldy

Haldy jsou specializované datové struktury, které splňují tzv. vlastnost haldy, tj. **pokud B je potomkem A, pak hodnota B >= hodnotě A v případě minimální haldy v případě max heap je to naopak.** Je to neefektivnější implementace prioritní fronty. V PALech řešila min heapa.

**Binární halda**

* téměř úplný binární strom = všechny hladiny jsou plně naplněné kromě poslední, která může a nemusí být plně naplněná
* strom splňuje vlastnost haldy
* reprezentována polem (na první pozici je kořen, 1. dítě je na 2\*n a druhé na 2\*n + 1, kde n je index nodu … pole je indexováno od 1)

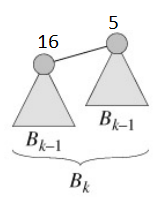
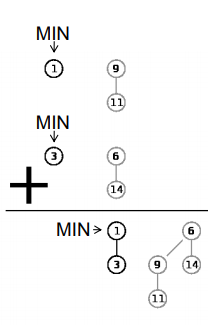


* **INSERT** [O(log(n)] - přidáme na konec haldy a poté swapujeme s rodiči, dokud nebude na správném místě.
* **ACCESS MIN** [O(1)] **-** minimum je v kořeni stromu
* **DELETE MIN** [O(log(n))] **(**[**YouTube**](https://www.youtube.com/watch?v=3Djx6T8xiy8)**)-** nastavíme do rootu hodnotu posleního uzlu v haldě a ten smažeme. Poté se podíváme na menšího ze dvou potomků a pokud je hodnota větší, swapneme uzly. Takto pokračujeme dokud nebude prvek na správném místě.
* **DECREASE KEY** [O(log(n))]**-** hodnotu v uzlu snížíme a poté podobně jako v INSERTU swapujeme s rodiči, dokud nebude na správném místě
* **DELETE** [O(log(n)] **-** funguje, jako delete min, akorát poslední uzel se neswapne s rootem ale s uzlem, který chceme smazat. Nebo můžeme snížit hodnotu na -∞ pomocí DECREASE KEY, čímž se dostane do kořene a pak použít DELETE MINp
* **BUILD HEAP -** Provádíme operaci insert dokud nevložíme všechny. Poté můžeme začít porovnávat od indexu N/2 k indexu 1 neboť tam se nachází poslední node, který má nějaké děti. U každého nodu zjistíme, jestli není porušeno pravidlo haldy, tedy jestli není nějaký z dětí menší. Pokud ano swapneme. Pokud jsou oba menší swapujemes tím menším z nich.

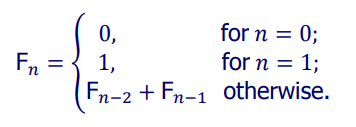
**D-nární halda**

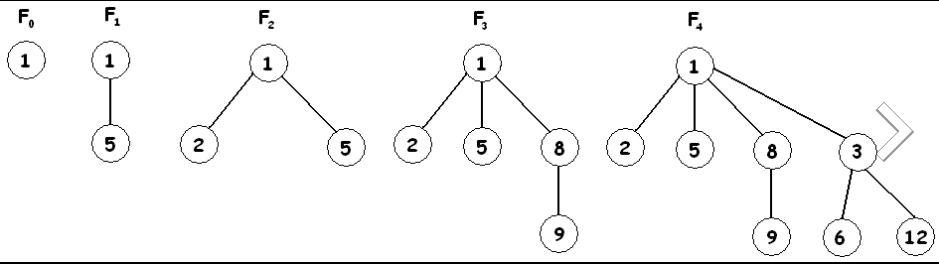
* Generalizace binární haldy, která má d potomků
* Všechno je analogické.
* Ve složitostech je místo dvojky déčko.
* Je vhodné kvůli efektivitě volit d jako mocniny dvojky kvůli bitovým posunům atp.
* D-nární halda je typicky rychlejší, pokud se halda nevleze do cache

**Binomiální haldy**

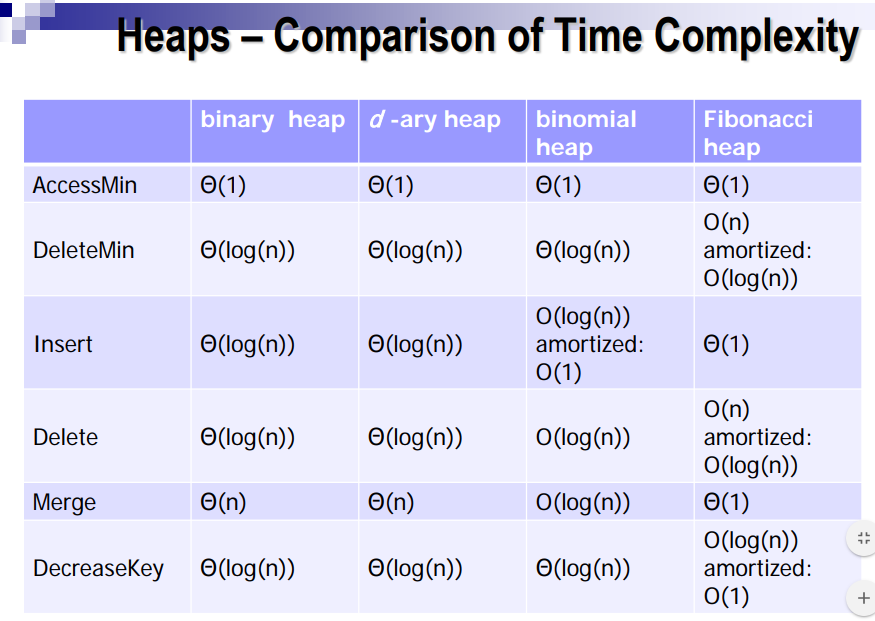
* kolekce binomiálních stromů stupně 0 až ⌊log(n)⌋.
* každý stupeň může být zastoupen pouze 0x nebo 1x
* každý binomiální strom splňuje vlastnost haldy (v tomto případě minimální)
* **Binomiální strom**
  + vlastnosti
    - splňuje vlastnost haldy
    - hloubka stromu je stejná jako stupeň stromu tj. k
    - kořen má tolik dětí, kolik je stupeň stromu
    - strom má 2k uzlů, kde k je stupeň stromu
    - v hladině i = 0, 1, …, k je přesně (ki) = k nad i uzlů
    - strom hladiny k má kořen, jehož potomci jsou kořeny bin. stromů stupně k-1, k-2, …, 0 (v tomto pořadí)
  + binomiální strom stupně k, lze sestavit ze dvou stromů stupně k-1 tak, že strom, který má v kořeni větší hodnotu se napojí na kořen druhého stromu jako nejlevější dítě.
* **INSERT** [O(log(n))] **-** vytvoříme novou binomiální haldu s jedním prvkem (tzn. jeden binomiální strom stupně 0) a mergneme ji s původní.
* **ACCESS MIN** [O(1)] - vrátí kořen binomiálního stromu z MIN pointeru 
* **MERGE** [O(log(n))]**-** zmerguje dvě haldy. Zmerguje binomiální stromy stejného stupně tak, že porovná hodnoty v kořenech stromů. Menší hodnota bude kořen a druhý strom bude jeho nejlevějším potomkem
* **DELETE MIN** [O(log(n)] - víme, který strom obsahuje minimum. Kořen smažeme, čímž se strom rozpadne na podstromy, které přidáme do nové haldy, z původní tyto stromy odebereme. Poté mergneme novou haldu s původní.
* **DECREASE KEY** [O(log(n))] - analogická operace jako u binární haldy (dějě se v rámci jednoho binomiálního stromu)
* **DELETE** [O(log(n))] - snížíme hodnotu toho, co mažeme na -∞ pomocí operace DECREASE KEY, čímž se dostane do kořene stromu a pak použijeme DELETE MIN

**Fibonacciho halda (**[**Vizualizace**](https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html)**)**

* modifikovaná binomiální halda - stromy nejsou seřazené v tom smyslu, že strom stupně k se skládá sice ze dvou stromů stupně k-1, ale strom může být jakýmkoliv potomkem kořene toho druhého (u binomiální haldy to musel být ten nejlevější) = nemusí to tvořit stále stejný pattern jako u binomiální haldy - struktura je flexibilnější
* v extrémním případě může každý prvek haldy tvořit izolovaný strom nebo naopak všechny prvky mohou být pouze v jednom stromu
* **POUŽITÍ -** není vhodná pro real time systémy, neboť některé z jejich operací (DELETE MIN, DELETE) mohou v nejhorším případě nabývat až lineární složitosti - což je moc.
* **Fibonacciho stromy**
  + stromy mají stupně podle fibonacciho posloupnosti, ]



* operace, které nejsou potřebné odkládáme a vykonáme je ve chvíli, kdy je to nevyhnutelné - např. ke spojování (merge) dochází často až při operaci Delete min nebo Decrease key
* **Pravidlo:** Lze odříznout nejvýše jednoho potomka od každého nekořenového prvku. Pokud je odříznut druhý potomek uzlu, musí být uzel odříznut od svého rodiče a stává se kořenem nového stromu.
* **Reprezentace:** 
  + obousměrný cyklický spojový seznam kořenů všech stromů - pro potomky každého prvku se vytváří obdobný seznam atd.
  + Každý uzel ví kolik má potomků
  + máme ukazatel na kořen, který obsahuje minimum haldy
* **ACCESS MIN** [O(1)] **-** máme na něj pointer
* **CONCAT (zřetězení)** [O(1)] **-** spojí se spojové seznamy s kořeny fib. stromů za sebe
* **CONSOLIDATION (slévání stromů)** [Amortizovaně O(log(n))]
  + sjednotíme kořeny stejných stupňů - kořen zůstane ten, který má menší hodnotu, aby se zachovalo pravidlo haldy, druhý bude jeho potomek (libovolný, nemusí být nejlevější jako u binom. haldy)
    - k hledání stromů stejného stupně se používá pomocné pole ukazatelů, ve kterém se uchovává reference vždy na jeden kořen každého stupně. Pokud se najde druhý strom stejného stupně, spojí se postupem definovaným výše a ukazatel v poli je aktualizován
* **DELETE MIN** 
  + odstraníme minimum, čímž se strom rozpadne na tolik částí, kolik má potomků, potomci minima se stanou kořeny nový fib. stromů.
  + teď musíme nálézt nové minimum - kořenů však může být po rozpadnutí stromů hodně a operace může být proto náročná, z toho důvodu se nejprve provede **slévání stromů**, čímž se počet kořenů zredukuje
  + zbývající kořenové prvky projdeme a nalezneme nové minimum, na kterého nastavíme ukazetel
* **INSERT** [O(1)]
  + vytvoří se nová halda s jedním stromem o jednom prvku a zřetězí (concat) se s původní haldou
* **DECREASE KEY** [Amortizovaně O(1)]
  + snížíme hodnotu a provedeme kontrolu pravidla haldy - tj. jestli je rodič stále menší než nová hodnota
  + pokud není, tak uzel, u kterého jsme hodnotu snížili, odřízneme a přidá se jako nový strom o jednom prvku.
  + pokud rodič, od kterého jsme ho odřízli není kořenem, **zvýrazníme ho**, přičemž pokud už **zvýrazněn byl**, odřízneme ho také, přidáme ho jako nový jednoprvkový strom a opět zvýrazníme jeho rodiče (toto se bude opakovat dokud nebude rodič neoznačený vrchol nebo bude kořenem)
  + prvky, které byly vloženy jako nové stromy a byly zvýrazněny, o zvýraznění přijdou
  + ***zvýrazňuje se kvůli výše zmiňovanému pravidlu***
* **DELETE** [Amortizovaně O(log(n))]
  + snížíme hodnotu klíče pomocí DECREASE KEY na **-∞ ,** tím se stane nejmenším prvkem haldy
  + poté použijeme operaci DELETE MIN k smazání minima haldy



# 3. Kombinatorické a číselně teoretické algoritmy, izomorfizmus, prvočíselnost

## 3.1 Generování a enumerace datových struktur a kombinatorických objektů

**Ranking function -** bijekce S -> {0, …, |S| - 1} tj. očíslování prvků konečné množiny od 0 po velikost množiny - 1.

* podle očíslování můžeme prvky množiny seřadit tj. následovník prvku **s bude prvek t** pokud rank(t) je rank(s) + 1
* lze využít k úspornějšímu ukládání kombinatorických objektů (ukládáme rank místo celého objektu)
* také lze využít k generování náhodných objektů z množiny s pravděpodobností 1/|S|

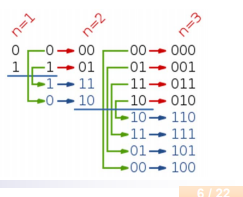
**Unrank** - opak ranku

**Subsets** (podmnožiny)

* možných podmnožim je 2N , kde N je velikost množiny
* **charakteristický vektor** podmnožiny (označme např. T)
  + jednodimenzionální binární pole x(T) = [xn-1, xn-2, …, x0]
  + říká jestli prvek n-i patří nebo nepatří do podmnožiny T (1 nebo 0)

**Grayův kód**

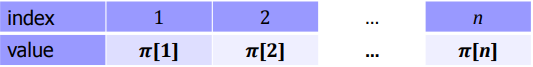
* binární kód, kde se dvě následující hodnoty liší vždy jen o 1 bit
* začínáme z [0 1], další ,,generaci” vytvoříme tak, že hodnoty zrcadlově překlopíme tj. [0, 1, 1, 0] a doplníme před první polovinu 0 a před druhou jedničky tj. [00, 01, 11, 10]



* **Vlastnosti:**
  + grayův kód se dá vytvořit z binárního pomocí vzorce **gj = (bj + bj+1) mod 2** tj. pro vypočtení pozice j sečteme bity z binárního kódu na pozici j a j +1 a modulíme 2
  + binární kód se dá vytvořit z grayova pomocí vzorce **bj = (gj + bj+1) mod 2** tj. pro vypočtení pozice j sečteme j-tý bit grayova a j+1 bit binárního a modulíme 2
  + binární kód se dá vytvořit z grayova pomocí vzorce **bj = (gi ) mod 2** tj. pro vypočtení pozije j sečteme j-tý až n-1 bit grayova kódu a modulíme 2

**K-prvkové podmnožiny**

* počet k prvkových podmnožin (kS) - s nad k
* reprezentace k prvkovým seřazeným jednodim. polem [t1 …. tk]

**Permutace**

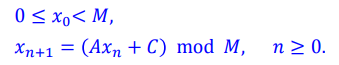
* bijekce z množniny sama do sebe
* reprezentace - jednodimenzionální pole

## 3.2 Generátory náhodných čísel

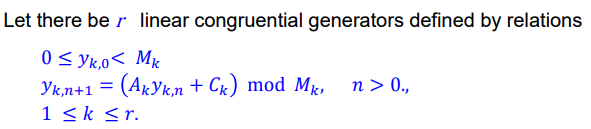
**Pseudonáhodný generátor**

* generuje náhodná čísla pouze zdánlivě
* po nějakém čase se začnou opakovat = perioda
* Dobrý generátor
  + má dlouhou periodu, čísla se neopakují tak často
  + generuje čísla z daného intervalu rovnoměrně
  + je rychlý - dobré jsou ty, které používají modulo mocninou dvojky, neboť bitové operace jsou rychlé

**Lineární kongruentní generátor (Linear Congruential Generator)**

* ****
* M je modulus, A je multiplier, C je inkrement, x0 je seed
* to, jak bude generátor dobrý záleží na navolení těchto hodnot
* určitě neplatí, že když zvolíme tato čísla jako prvočísla, bude generátor dobrý
* K dosažení maximální možné délky periody musí čísla splňovat následující (**Hull-Dobell theorem)**:
  + C a M jsou nesoudělná (největ. spol. dělitel [GCD] = 1)
  + A-1 je dělitelné všemi čísly z prvočíselného rozkladu čísla M
  + pokud je M dělitelné 4 pak je také A-1 dělitelné 4

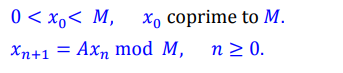
**Kombinovaný lineární kongruentní generátor (Combined Linear Congruential generator)**



* zkombinujeme r linearně kongruentních generátorů
* maximální délka periody je 

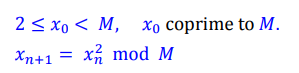


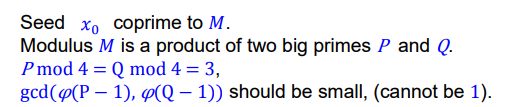
**Lehmer generator**



* M je modulus, x0 je seed, a A je multiplier
* Délka periody je maximální (rovna M-1) pokud je
  + M prvočíslo
  + A je primitivní kořen multiplikativní skupiny celých čísel modulo M
    - multiplikativní skupina je množina {A, A2, A3, … , AM-1}
    - každý člen zmodulíme M

**Blum Blum Shub Generator**

****

* M je modulus, x0 je seed
* 

## 3.3 Prvočísla

### 3.3.1 Pojmy

**Prime counting function π(n)**

* spočítá počet prvočísel menších nebo rovných n

**Euler’s totient function φ(n)**

* spočítá počet kladných celých čísel menších nebo rovných n, která jsou nesoudělná s n

**Mersennovo prvočíslo**

* takové prvočíslo, které je o 1 menší než celočíselná mocnina dvojky tj. Mp = 2p - 1
* napříkald 7 = 23 - 1
* v tomto případě lze ukázat, že i exponent, na který 2 umocňujeme je prvočíslo

### 3.3.2 Algoritmy pro prvočísla

**Eratosthenovo síto**

* generuje prvočísla do daného čísla n
* jdeme po číslech od 2 výše a vždy ze síta vyškrtneme všechny jeho násobky. Toto stačí dělat do odmocniny z n
* násobky v iteraci i stačí vyškrtávat od i2, i2+i, i2+2i, … dokud je to menší než n
* O(n log log n)

**Malá fermatova věta**

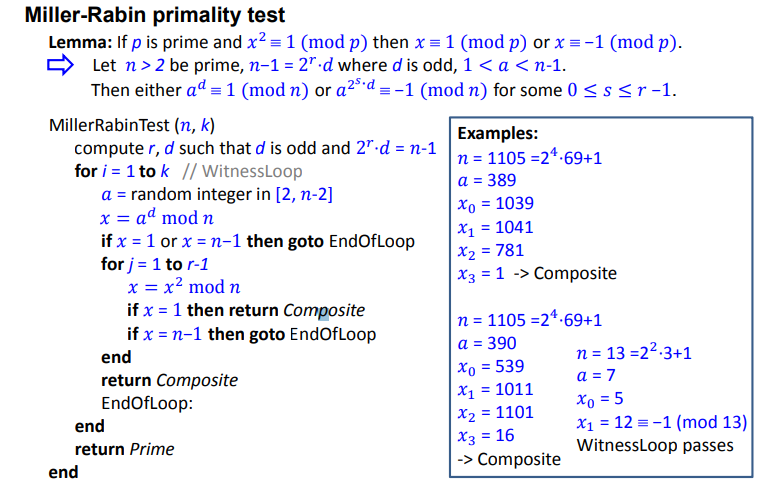
* pokud je číslo p prvočíslo a máme číslo a, které je 0 < a < p, potom ap-1 je kongruentní s 1 modulo p: ap-1 ≡ 1 mod p (pre debilov ap-1 mod p = 1)

**Fermatův test prvočíselnosti**

* **patří do skupiny náhodných testů prvočíselnosti (Randomized primality test)**
* vstup čísla n, k
* od 1 do k
  + a = náhodné číslo z intervalu <2, n-2>
  + pokud neplatí malá fermatova věta potom se jedná o číslo složené
* pokud projde for tak test vrací, že je to prvočíslo, což ještě nemusí být pravda (je to ale velmi pravděpodobné)

**Miller-Rabin**

* **patří do skupiny náhodných testů prvočíselnosti (Randomized primality test)**
* časová složitost O(klog3n)
* chybovost algoritmu (false positive) je nejvýše 4-k



**AKS test prvočíselnosti**

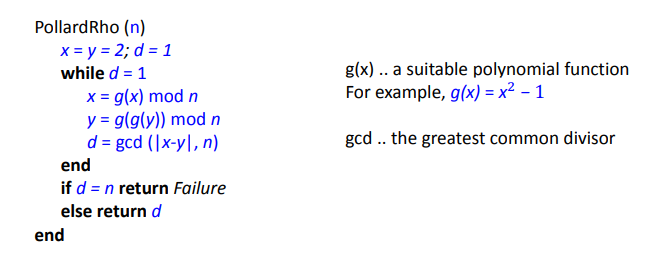
* první deterministický (ne pravděpodobnostní) algoritmus na test prvočíselnosti pracující v polynomiálním čase
* O(log6n)
* obrovský teoretický význam, ale nepoužívá se v praxi

### 3.3.3 Algoritmy pro faktorizaci celých čísel

* neexistuje žádný efektivní algoritmus

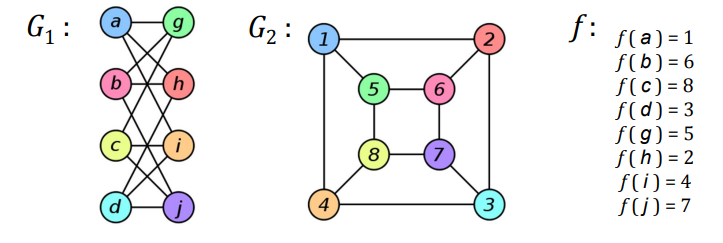
**Pollard’s rho algorithm**

* **efektivní pro** složená čísla, která mají malý prvočíselný rozklad - očekávaná složitost O(n1/4)
* pseudokód níže

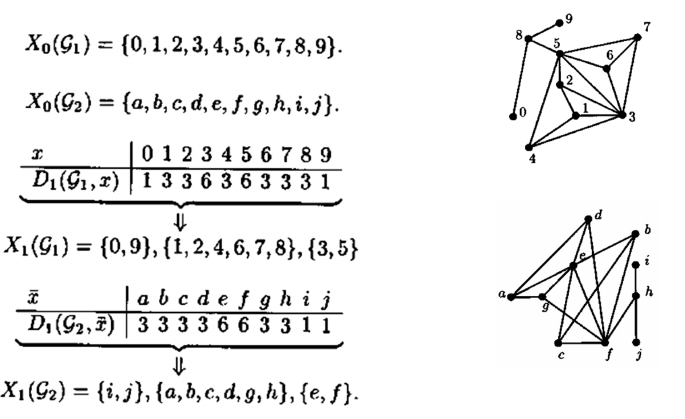


## 3.4 Izomorfismus

,,Dva grafy jsou izomorfní pokud mají stejný počet vrcholů a vrcholy, které jsou v jednom grafu spojeny hranou jsou v druhém také spojeny hranou”



**Invariant** - funkce, která dává stejnou hodnotu pro grafy, které jsou izomorfní

* například |V| je invariant - pokud jsou dva grafy izormorfní, musí mít stejný počet vrcholů

**Invariant inducing function**

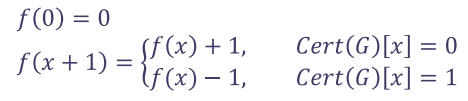
* funkce, která ,,vede k invariantu”
* např. stupeň vrcholu
* v příkladu výše se vytvoří skupiny vrcholů, které mají stejný stupeň v obou grafech
* ve skupinách už jde vidět, které skupiny by se mohly na sebe namapovat. To však nemá jednoznačné řešení - je zde celkem (2!)(6!)(2!) = 720 možností jak na sebe grafy namapovat (čísla ve vzorci jsou určena podle velikosti skupin)
* ne každé mapování musí být izomorfismus! Platí to jenom pro některá možnosti

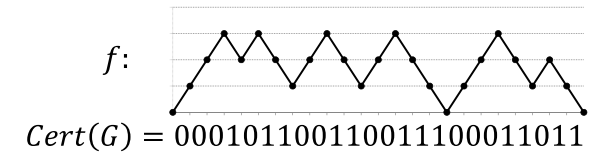
**Certifikát grafu**

* dva certifikáty jsou shodné, právě tehdy když jsou grafy izomorfní
* používaný nejrychlejšími algoritmy pro výpočet izomorfismu
* lze použít na obecných grafech, ale i na speciálních skupinách grafů jako jsou např. stromy

**Certifikát stromu**

* sekvence 0 a 1
* Vlastnosti:
  + počet 1 a 0 je stejný
    - a to u každého vrcholu v každé chvíli algoritmu
  + délka je rovna počtu vrcholů krát 2 (2\*|V|)
* Postup:
  + označíme všechny uzly 01
  + dokud jsou v grafu více než dva uzly provádíme následující:
    - pro každý uzel který **není** list nahradíme jeho certifikát certifkátem **listů, které s ním sousedí** (tyto certifikáty za sebe spojíme v lexikografickém pořadí) a přilepíme na začátek 0 a na konec 1
    - tyto listy, které s ním sousedí odmažeme
  + pokud nám na konci zůstane jen jeden uzel bude v něm certifikát grafu
  + pokud nám na konci zůstanou dva uzly spojíme jejich řetězce v lexikografickém pořadí a máme certifkát
  + příklad v [přednášce](https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/pal07.pdf)
* Z certifikátu lze strom rekonstruovat:



* + začínáme na 0
  + za každou 0 jdeme o 1 výše
  + za každou 1 jdeme o 1 níže
  + dostaneme něco takovéhohle
    - jdem odspodu nahoru - každý level je rozdělen na několik částí
    - to udává kolik potomků uzel má
    - v příkladu máme 2 spojené kořeny - 1. má pak 3 potomky, 2. má 2 potomky
    - 1. z těch 3 potomků prvního kořene má 2 potomky, 2. má jednoho, 3. má jednoho
    - 1. z těch 2 potomků druhého kořene má 1 potomka, 2. nemá žádného

# 4. Vyhledávací stromy a jejich využití

### 4.1 BVS (BST) - Binary search tree

<https://www.youtube.com/watch?v=wcIRPqTR3Kc>

**Vlastnost BVS -** hodnoty v levém podstromě uzlu jsou vždy menší a hodnoty v pravém podstromě jsou vždy větší než hodnota uzlu

**Reprezentace** - jednodim. pole

**Vlastnosti:**

* není většinou vybalancovaný
* není regulární - tj. každý uzel, který není list má stejný stupeň. TO není pravda neboť uzly nemusí mít vždy dva potomky, někdy mohou mít jen jednoho
* pokud projdeme BVS inorder, tak dostaneme seřazené pole jeho hodnot

**Operace**

* FIND
  + začneme v kořeni
  + pokud je hledaná hodnota menší jdeme doleva
  + pokud je hledaná hodnota větší jdeme doprava
  + pokud dojdeme do listu a list není onou hodnotou, hodnota ve stromu není
  + O(log(n)) když je vybalancovaný, až O(n) když není
* INSERT
  + najdeme jako ve findu místo, kam by měl být prvek vložen, tzn. dojdeme do listu a pokud je hodnota vkládaného prvku větší, připojíme ho jako pravého potomka, pokud menší jako levého potomka
  + Θ(log(n)) když je vybalancovaný, až O(n) když není
* DELETE
  + komplikovanější - záleží jestli mažeme list nebo vnitřní uzel s jedním potomkem, nebo vnitřní uzel se dvěma potomky
  + O(log(n)) když je vybalancovaný, až O(n) když není
* DELETE - LIST
  + najdeme ho jako ve find a prostě odmažeme
  + O(log(n))
* DELETE - VNITŘNÍ UZEL 1 POTOMEK
  + uzel se najde FIND operací
  + uzel se vymaže a na jeho rodiče se naváže potomek mazaného uzlu
* DELETE - VNITŘNÍ UZEL 2 POTOMCI
  + uzel se najde FIND operací
  + před smazáním najdeme **nejlevější** prvek v pravém podstromě nebo **nejpravější** prvek v levém podstromě.
  + tímto prvkem nahradíme mazaný prvek - tj. ten který mažeme odstraníme a náhradník dostane reference na jeho potomky a rodiče a odstraníme ho z původního místa
  + při odstranění je možné, že nastane situace DELETE - list nebo DELETE s jedním potomkem

### 4.2 AVL

**Vlastnosti**

* samovyvažující se strom
* je téměř vybalancovaný - rozdíl hloubek levého a pravého podstromu každého uzlu ve stromě je maximálně 1
* vyvážení se dělá pomocí rotací
* díky tomu, že je AVL téměř vybalancovaný jsou složitosti operací FIND - O(log(n)), INSERT - Θ(log(n)) a DELETE - Θ(log(n))

**Operace**

* operace se dějí stejně jako v BVS, ale vždy, když něco uděláme Insert nebo Delete, jdeme od modifikovaného uzlu po rodičích směrem do kořene a updatujeme hloubky podstromů. Pokud dojde k rozvážení tj. rozdíl hloubky levého a pravého podstromu je vetší než 1 musí se **provést jedna z rotací.** Která rotace se provede záleží na tom, jak jsme do rozváženého uzlu přišli:
* R - pravá rotace - 2x doprava nahoru
  + tj. levý podstrom je rozvážený
  + prvek který to způsobuje je v levém podstromě levého potomka rozváženého uzlu
* L - levá rotace - 2x doleva nahoru
  + tj. pravý podstrom je rozvážený
  + prvek, který to způsobuje je v pravém podstromě pravého potomka rozváženého uzlu
* LR - levo pravá rotace - 1x doleva nahoru a 1x doprava nahoru
  + tj. levý podstrom je rozvážený
  + prvek, který to způsobuje je v pravém podstromě levého potomka rozváženého uzlu
* RL - pravolevá rotace
  + tj. pravý podstrom je rozvážený
  + prvek, který to způsobuje je v levém podstromě pravého potomka rozváženého uzlu

**Postup rotací**

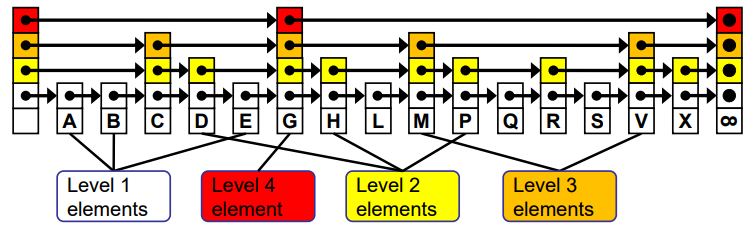
* R rotace - levý potomek (např. A) přijde na místo rozváženého uzlu (např. C). C pak bude pravým potomkem uzlu A. Pravý podstrom uzlu A bude nyní levým podstromem uzlu C
* L rotace - analogicky k R rotaci akorát naopak
* LR rotace - udělá se levá rotace v levém potomkovi rozváženého uzlu a pak pravá rotace v rozváženém uzlu
* RL rotace - udělá se pravá rotace v pravém potomkovi rozváženého uzlu a pak levá rotace v rozváženém uzlu
* **ACHTUNG: při INSERTU stačí udělat jedna z rotací a víme, že je vyvážený, při DELETU to platit nemusí a musí se zkontrolovat všechny uzly až ke kořeni.**

### 

### 4.3 Skip list

**Info**

* hledání v klasickém linked listu prvek po prvku je pomalé
* seřazený linked list, kde každý uzel obsahuje jeden či více odkazů na další prvky v listu v různé vzdálenosti - tj. umožňuje přeskakovat a tím zrychlit vyhledávání



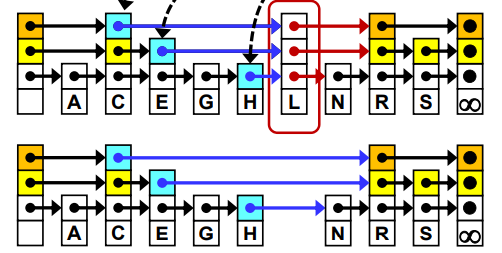
* na konci (nebo na začátku při kruhové implementaci) skip listu musí být zarážka (sentinel), která zajistí, že nebudeme skákat (skipovat) přes hranice listu)

**Určení levelu prvku**

* level prvku se určuje náhodně
* např. hodem mincí - dokud padá hlava zvyšujeme level, pokud padne orel nebo jsme dosahli maximálního možného levelu končíme.
* použitím generátoru pseudonáhodných čísel (např. Lehmer generator)
* obecně jde o to vybrat s nějakou pravděpodobnost **p** (mezi 0 a 1)
  + v průměru pak 1-p prvků mívá level 1, (1-p)2 prvků level 2, (1-p)3 level tři atd.
    - v případě mince je to p = 0,5, což nám dává polovinu prvků levelu 1, čtvrtinu prvků levelu 2, osminu prvků levelu 3 atd.
  + **od zvoleného p** se odvíjí **efektivita skip listu**
  + bylo zjištěno empiricky, že nejvhodnější je zvolit p = 0.25
    - čas vyhledávání je téměř stejný, ale v ušetříme pamět na pointrech (kterých je v průměru **N \* 1/(1-p)**).

**Operace**

* FIND (SEARCH)
  + začneme ve vrchním levelu (tj. ten to přeskakuje nejvíce)
  + podíváme se na další prvek a jestli je menší než to co hledáme, tak si vesele skočíme
  + pokud není menší jdeme o level níže a opakujeme proces
  + takto to děláme, dokud prvek nenajdeme nebo dokud dojdeme k zarážce na nejnižším levelu, což znamená, že tam prvek není.
* INSERT
  + findem najdeme místo, kam by se měl prvek vložit
  + určíme level prvku - házení mincí (viz specialni sekce)
  + vložíme prvek a všechny jeho levely postupně odspodu - ke správnému navázání pointerů prvků před a po, se často používá pomocné pole
* DELETE
  + findem prvek najdeme
  + pak ho vymažeme a za pomocí pomocného pole opět přepošleme odkazy, které na něj mířili na další prvek
  + implementační zrychlení: pokud odkaz předka nemířil na tento prvek tj. předek byl levelu 3 a mazaný prvek jen levelu 2, v momentě přesměrování odkazů končíme, když dosáhneme levelu 2.)
* Všechny mají složitost O(log(n))
  + ukazuje se to komplikovaně, neboť je do toho zamíchaná ta pravděpodobnost



### 4.4 B a B+

#### 4.4.1 B tree

**Vlastnosti:**

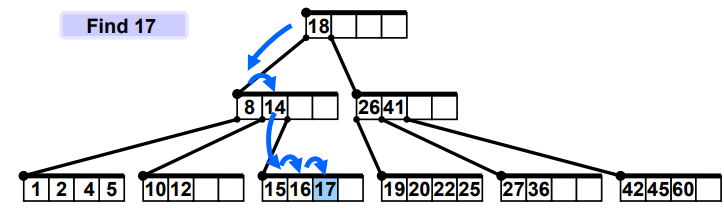
* vyvážený - roste odspodu nahoru
* každý node obsahuje více hodnot - to je určeno pomocí parametru k > 1,
  + node, který není list nebo kořen musí obsahovat alespoň k hodnot a nejvíce 2k hodnot
  + tzn. každý takový node má alespoň k+1 a nejvíce 2k+1 potomků
  + kořen smí obsahovat 1 až 2k hodnot, takže má nejméně 2 a nejvíce 2k+1 potomků
* **alternativní definice** - řád stromu
  + tj. něco jako k, ale opačně, pokud má b-tree řád **b**, tak všechny uzly kromě kořene mají maximálně **b** a minimálně **ceil(b/2)** potomků.
* hodnoty v uzlu jsou seřazené

**Updatovací strategie**

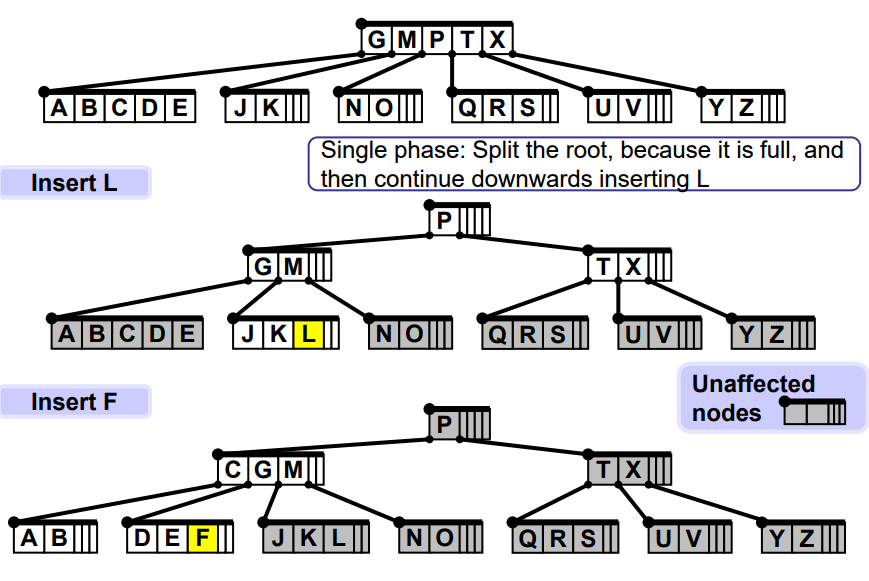
* multiphase - vše se dělá na několik fází - dělení uzlů se provádí až když je to nevyhnutelné
* single phase - vše se udělá v jedné fázi - dělení uzlů se provádí předem

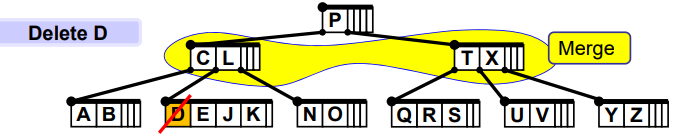
**Operace**

* **FIND**

- 

* **INSERT (multiphase strategy)**
  + najdeme místo pro prvek pomocí FIND
  + pokud je v uzlu místo, vložíme a jsme v cajku
  + pokud není místo:
    - seřadíme hodnoty uzlu včetně vkládané hodnoty mimo strom
    - vybereme medián (prostřední prvek), který bude novým dělícím klíčem, který vložíme do rodiče
    - levý potomek pak bude mít levou polovinu hodnot, pravý potomek pravou polovinu hodnot
    - pokud je rodič plný a nelze tam vložit nový dělící klíč, proces se opakuje tak dlouho dokud to nepůjde - může vyústit v to, že vznikne nový kořen a výška stromu se zvětší o 1
* **INSERT (single phase strategy)**
  + narazíme-li při insertu (tj. při hledání místa kam prvek vložit) na plný node, rovnou ho splitneme, tak jak je popsáno výše



* **DELETE (multiphase)**
  + při deletu mohou nastat tyto situace
    - mažeme z dostatečně plného listu
    - mažeme z vnitřního nodu
    - mažeme z nedostatečně plného listu
      * ale sousední list je dostatečně plný a lze mu ukrást jeden prvek
      * ale sousední list není dostatečně plný
  + **DELETE z dostatečně plného listu**
    - easy shit, prostě ho najdeme a smažeme
  + **DELETE z vnitřního nodu**
    - je potřeba prvek substituovat
    - podobně jako v BVS najdeme nejlevější uzel v pravém podstromu (a vezmeme jeho první hodnotu) nebo nejpravější prvek v levém podstromu (a vezmeme jeho poslední hodnotu)
  + **DELETE z nedostatečně plného listu - sousední list je dostatečně plný**
    - prvek se smaže
    - zbylé prvky dáme dohromady s prvky dostatečně plného sousedního listu a s dělícím klíčem z rodiče
    - toto sestortujeme mimo strom -> vybereme medián tj. nový dělící klíč do rodiče -> hodnoty nalevo do levého potomka -> hodnoty napravo do pravého potomka
  + **DELETE z nedostatečně plného listu - sousední listy nejsou dostatečně plné**
    - hodnotu smažeme
    - dáme podobně jako výše dohromady zbylé prvky s prvky ze souseda a s dělícím klíčem z rodiče -> sesortujeme -> vložíme do uzlu, ze kterého jsme mazali
    - sousední uzel je teď prázdný, takže ho celý smažeme
    - smažeme dělící klíč z rodiče, neboť teď je součástí hodnot v uzlu, ze kterého jsme mazali
    - toto může porušit pravidlo B stromu v rodiči - aplikujeme stejný proces na rodiče - což může vést ke snížení výšky stromu o 1
* **DELETE (singlephase)**
  + pokud mažeme z vnitřního nodu a žádný z potomků nemá klíč nazbyt, za který by ho šlo substituovat, potomci se před deletem mergnou do jednoho nodu
  + pokud mažeme z listu a uzly o level výš už mají jen tak akorát klíčů (tzn. že by delete z nich způsobil v multiphase nějaký merge), provedeme merge rovnou dopředu
  + ****

#### 4.4.2 B+ tree

**Vlastnosti:**

* upgradovaný B tree
* hodnoty ve vnitřních uzlech fungují jako směrovače tj. směrují v kterém listu se nachází hodnota, resp. pointer na danou hodnotu v paměti
* směrovače mohou ve stromě zůstat i když hodnota již byla smazána
* použití v databázích
* dolní hladina stromu, kde jsou listy je propojená a tvoří linked list - to je kvůli efektivitě, aby mohly být některé hodnoty získávány (například v dotazech i chceme získat nějaký interval či oblast hodnot) i bez toho, abychom museli projít celým mechanismem B+ stromu.

Složitost Find, Insert a Delete je Θ(b\*logb(n)) - n je počet záznamů ve stromě a b je řád stromu (viz definice b stromu)

**Operace**

* **INSERT**
  + 3 případy:
    - v listu je místo
    - v listu není místo, ale v rodiči místo je
    - v listu není místo, v rodiči taky ne
  + INSERT - v listu místo je
    - pohoda prostě to tam vlož
  + INSERT v listu není místo, ale v rodiči jo
    - stejně jako v B tree - sesortuj hodnoty v listu včetně vkládané hodnoty, vyber medián a ten vlož do rodiče
    - v levém potomkovi budou hodnoty menší než medián
    - v právém potomkovi budou hodnoty vetší **nebo rovno** než medián (to je ten rozdíl oproti B tree)
  + INSERT v listu není místo v rodiči taky ne
    - podobně jako v případě výše (pamatovat na větší nebo rovno!)
    - rodič je plný, musíme ho taky splitnout - zase stejný princip a zase budou v pravém potomku mediánu hodnoty větší **nebo rovno**
    - atd. než dojdeme do situace, že medián do rodiče vložit půjde
* **DELETE**
  + tady dává smysl jen delete v listu, neboť listy jediné míří na nějakou skutečnou hodnotu (tzn. nejsou jen směrovače)
  + podobně jako v B-tree 3 případy
    - list má dostatečný počet hodnot
    - list nemá dostatečný počet hodnot, ale někdo ze sousedů má
    - list nemá dostatečný počet hodnot a ani žádný soused
  + vše se děje analogicky s tím rozdílem, že když smažeme hodnotu z listu, která je přítomna také jako směrovač v bezprostředním rodiči, tak směrovací hodnotu nahradíme prvním větším prvkem ze zbývajících prvků (viz [strana 38](https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/paska11b.pdf))

### 4.6 R-B tree

**Vlastnosti**

* červeno černý BST - nody si drží info o své barvě
* listy mají dva **černé** nil nody - pointery na neexistujícího potomka
* zhruba vybalancovaný - platí, že jeho výška je <= 2\* výška vybalancovaného stromu
* Black-height(x) - počet černých nodů na cestě z uzlu x do listu (uzel x nepočítáme, nil počítáme)
  + bývá někde mezi polovinou výšky stromu a výškou stromu

**Pravidla**

* nody musí být jen **červené** nebo **černé**
* **nil** nody jsou **černé**
* pokud je node **červený**, oba jeho potomci jsou **černí**
* každá cesta z nodu do nějakého potomka má stejný počet **černých** nodů - tedy black height je pro všechny cesty stejná
* kořen je **černý**

**Operace**

* **FIND** - jako v **BST** - barvy operaci neovlivňují
* **INSERT** 
  + obarvíme nový node na **červeno**
  + vložíme jako v klasickém BST
  + pokud je rodič vloženého nodu **černý** - nemáme problém - insert done
  + pokud je rodič vloženého nodu **červený** - 3+3 případy
  + Insert stojí Θ(log(n)) a jsou provedeny maximálně 3 rotace

### 

* **DELETE (na inda se mrkněte** [**ZDE - YouTube**](https://www.youtube.com/watch?v=CTvfzU_uNKE) **)**
  + zase jako v normálním BST
  + pak může nastat 6 případů
  + delete stojí Θ(log(n)) a jsou provedeny maximálně 3 rotace

### 

### 

### 4.7 Splay tree

**Vlastnosti**

* binární vyhledávací strom
* pokaždé, když přistoupíme nebo vložíme novou hodnotu, tak “přesuneme” node do kořene
* přesunutí (splay) se děje pomocí spec. rotací
* složitost všech operací je O(n) amortizovane však O(ln(n)) - složitost je těžké ukázat

**Rotace**

* jaká rotace se provede záleží, jak jsme do uzlu přišli podobně jako u AVL
* ZIG rotace
  + stejná jako levá nebo pravá rotace v AVL stromu
* ZIG-ZIG rotace
  + stejná jako bychom udělali R R nebo L L rotaci v AVL stromu
* ZIG-ZAG
  + stejná jako bychom udělali L R nebo R L rotaci v AVL stromu

**Operace**

* INSERT A FIND - jako v BST až na to, že se poté node přesune do kořene pomocí rotací
* DELETE
  + provede se FIND - to přesune node do kořene
  + smažeme root - rozdělí strom na levý a pravá podstrom
  + FIND maximum v levém podstromě - nejpravější prvek - toho přesune do kořene levého podstromu
  + navážeme pravý podstrom na nový kořen

**Výhody**

* amortizovaná složitost je strovnatelná s AVL nebo RB stromem
* implementace je jednodušší není třeba držet si další info jako je barva nebo výška

**Nevýhody**

* strom se mění i jen při read-only operacích

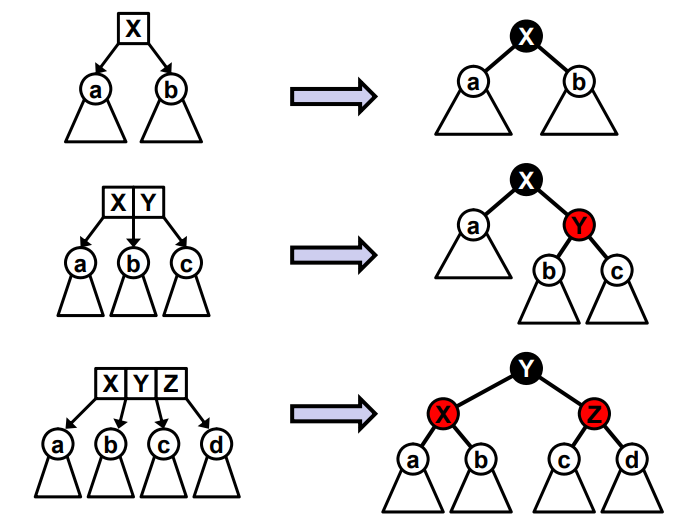
### 4.8 2-3-4 strom

**Vlastnosti**

* V podstatě B strom s minimálním stupněm 2 a maximální 4 tj. existují zde 3 typy nodů - node se 1 hodnotou (2-node), node se 2 hodnotama (3-node) a node se 3 hodnotama (4-node)
* tzn. ze mají vždy 2, 3 nebo 4 potomky (kromě listů samozřejmě)
* je perfektně vybalancovaný - všechny listy mají stejnou vzdálenost od kořene

**Operace**

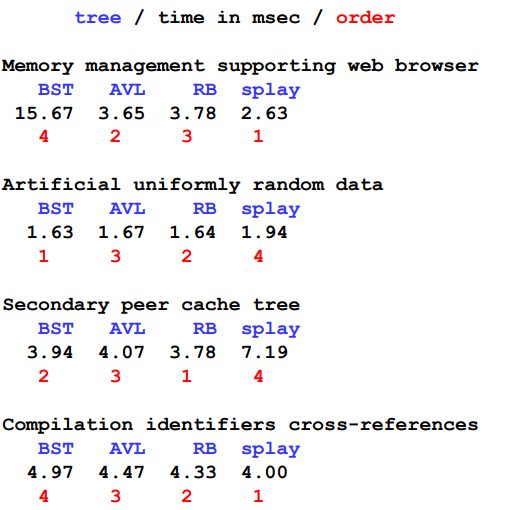
* FIND - jako v B stromě
* INSERT - jako v B stromě, akorát splitovací strategie (pokud je potřeba) se trochu liší
  + když hledáme místo, kam prvek vložit, tak pokaždé když narazíme na 4-node, splitneme ho na dva 2-nody - medián vložime do rodiče.
    - pokud vznikne 4 node vložením mediánu do rodiče, tak se rekurzivně splitne taky
  + to je podobné jako v single phase insertu v B tree - díky tomu není třeba po insertu splity řešit na cestě zpátky ke kořeni - to je prý časově náročné
* DELETE - jako v B-tree



spojitost 2-3-4 stromů s R-B stromy

### 4.9 Porovnání stromů

* u neseřazeného inputu jsou lepší nevybalancované BST
* pokud náhodný input obsahuje seřazené části - R-B tree
* pokud se do stromu vkládá v seřazeném pořadí -
  + pokud se poté přistupuje k hodnotám náhodně - AVL
  + pokud se poté přistupuje k hodnotám sekvenčně - Splay tree
* Použití v reálném prostředí
  + BST - jádra operačních systémů - sledování oblastí virtuální paměti
    - Linux < 2.4.10 - AVL
    - OpenBSD a Linux > 2.4.10 - R-B tree
    - FreeBSD, Windows NT - Splay tree
  + B tree - databáze



### 4.8 K-D strom - hledání ve více dimenzích

**Definice**

* BVS, který slouží k ukládání a vyhledávání v k-dimenzionálních datech
* v každém uzlu je k prvků, které určují pozici bodu v prostoru
* strom rozděluje k-dimenzionální prostor na obdélníkové oblasti - s každým uzlem ve stromě jsou asociovány dvě hyperplany (pokud k=2 tak to jsou obdélníky)
  + levému podstromu odpovída hyperplana, která obsahuje menší prvky
  + pravému podstromu odpovídá hyperplana, která obsahuje prvky, které jsou vetší nebo rovno
  + uzel v podstatě rozsekne prostor tvarem, který je o jednu dimenzi nižší - tj. například trojrozměrný prostor rozsekne na dva dvojrozměrnou rovinou, dvojrozměrný prostor jednorozměrnou přímkou atp.
  + dimenze ve které se řeže se střídá podle hloubky stromu
* díky tomu se dá ve stromě vyhledávat snadno podle pozice v prostoru

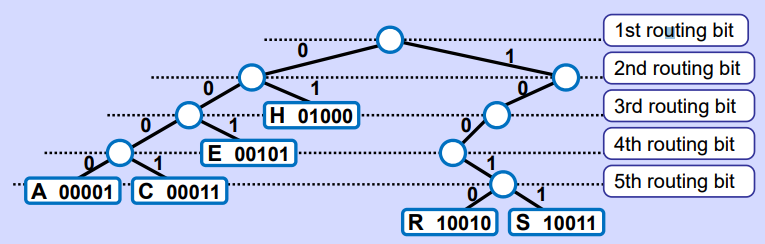
### **Operace**

* **FIND** - jako v normálním BVS
  + musíme si dát pozor v které jsme dimenzi
* **INSERT** 
  + najde se (FIND) kam uzel patří a uzel se vloží - podle toho do jaké hloubky jsme ho vložili víme, kterou dimenzi by měl rozdělit - po insertu prostor rozdělíme
* **FIND MIN (dim = k)**
  + hledá prvek, který má k-tý prvek (k-tou souřadnici) minimální
  + součástí DELETE operace
  + K-d strom si neudržuje informaci o žádném minimu, je nutné ho pracně najít - nejdražší operace v k-d stromu
    - O(n1 - 1/k) při n nodech a dimenzi k, pro k=2 je to O(n1/2)
  + Pokud třeba hledáme minimum souřadnice y v K-d stromu dimenze 2 - tj. FIND MIN (dim = y) víme, že když budeme v hloubce odpovídají dimenzi x musíme prohledat oba podstromy, ale když budeme v hloubce odpovídající dimenzi y, stačí prohledat pouze levý podstrom
* **DELETE**
  + při delete operaci se fyzicky mažou jen listy stromu
  + pokud mažeme vnitřní node, tak dojde k substituci za jiný, který je hlouběji ve stromu - pokud je to zase vnitřní uzel, rekurzivně substituce, pokud list, tak smažeme - nakonec to bude vždycky list
  + Postup
    - chceme smazat uzel X
    - pokud je pravý podstrom X **neprázdný**
      * najdeme v **pravém** podstromu pomocí operace FIND MIN prvek, který má minimum ve stejné dimenzi jako je prvek, který mažeme (tzn. mažeme-li z hloubky která odpovídá dimenzi např. y, hledáme prvek, jehož y-tá souřadnice je minimální)
      * mazaný prvek nahradíme nalezeným minimálním prvkem a minimální prvek rekurzivně smažeme
    - pokud je pravý podstrom X **prázdný**
      * najdeme v **levém** podstromu pomocí operace FIND MIN prvek, který má minimum ve stejné dimenzi jako je prvek, který mažeme (tzn. mažeme-li z hloubky která odpovídá dimenzi např. y, hledáme prvek, jehož y-tá souřadnice je minimální)
      * mazaný prvek nahradíme nalezeným minimálním prvkem a přesuneme prvky levého podstromu mazaného prvku do pravého podstromu
      * minimální prvek rekurzivně smažeme
* **Hledání nejbližšího souseda**
  + začínáme v kořeni, hledáme v obou podstromech
  + držíme si informaci o bodu, který byt zatím nejblíže tomu co chceme - co to je za bod a jakou má euklidovskou vzdálenost (vzhledem k našemu požadavku)
    - podle této informace lze hledání zařezávat
  + existují různé heuristiky na to, který podstrom prohledat dříve a jak dojít co nejrychleji k výsledku
    - v praxi prý dost pomáhá jít do toho podstromu, jehož rodič má lepší euklidovskou vzdálenost k planě, ve které se nachází bod, kterému chceme nalézt nejbližšího souseda
    - pokud je vzdálenost rodiče podstromu od plany, ve které se nachází bod, ke kterému chceme nalézt nejbližšího souseda, **větší než aktuální minimální vzdálenost**, můžeme podstrom zaříznout
  + složitost:
    - nejhůře to může být O(n), ale to pouze když je uspořádání bodů nešťastné tj. např. když:
      * je dimenze relativně velká tj. > 7
      * uspořádání bodů v nízké dimenzi je velmi speciální - např. uměle vykonstruované
    - pro rovnoměrně rozložená data bývá složitost lepší O(2Dimenze + logn)
      * efektivní pouze když je číslo 2Dimenzi o dost menší než **n**

### 4.4 Binary trie

**Charakteristika**

* klíče (hodnoty) mají kromě hodnoty taky svou sekvenci bitů, podle které se určuje poloha ve stromě - délka sekvence je u všech listů stejná - odpovídá největší hloubce stromu
  + díky tomu je zajištěno, že žádna sekvence v listu není prefixem sekvence jiného listu
* klíče jsou uloženy pouze v listech, vnitřní nody slouží jako směrovače (routery)
  + z každého vnitřního nodu vede 1 nebo dvě cesty do potomků - pravý potomek odpovídá bitu 1, levý potomek bitu 0
  + sekvence bitů v listu určuje cestu, jak jsme se do něj z kořene dostali
  + **h-tý bit** v listu určuje, jestli se ve vnitřním nodu v hlobce **h** šlo doprava nebo doleva
  + pokud vnitřní uzel nemá jednoho z potomků **nazývá se t**o **null link -** pokud nemá strom null linky je **perfektně vyvážený** a obsahuje 2h+1 - 1 uzlů a 2h listů
  + obecně je strom dobře vyvážený, ale obsahuje často spoustu zbytečných routovacích uzlů



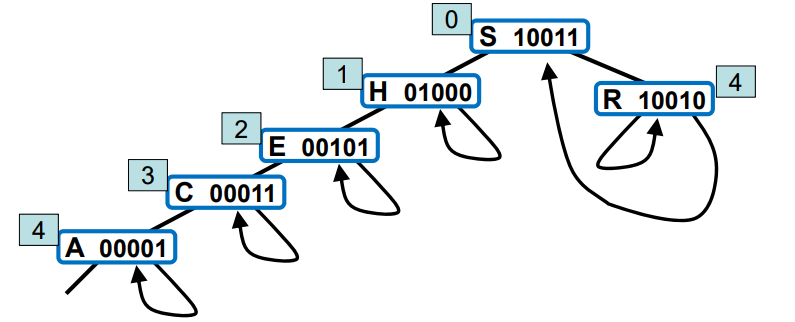
**Operace**

* INSERT
  + přidáním nového uzlu může vzniknou spoustu nových vnitřních směrovacích uzlů - vytvoření směrovacích uzlů je **,,lazy”**
  + při přidávání se vždy řeší jen bit v dané hloubce - například když do prázdného stromu vložíme 00001, tak to neznamená, že hned vytvoříme strom o hloubce 5, ale stačí podle prvního bitu, což je 0, prvek pověsit doleva. Až vložíme něco co bude začínat zase 0 a bude potřeba cestu do těchto dvou nodů odlišit, vyrobí se nové příslušné nové směrovací uzly - viz strana 3 v [přednášce](https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/paska13trie.pdf).
* SEARCH
  + pro určení kam ve stromě při hledání jít se používají bity - 0 do levého podstromu, 1 do pravého podstromu
  + pokud dojdeme do listu musíme ještě checknout jestli node obsahuje skutečně hodnotu, kterou hledáme (může se stát, že node který hledáme bude mít stejnou routovací sekvenci bitů ale jinou hodnotu)
* DELETE
  + opak k insertu - v přednáškách se nevysvětluje, ale očekával bych, že se hodnota najde, smaže a pak se odmažou nepotřebné routovací vnitřní uzly

### 4.5 Patricia trie

**Charakteristika**

* Patricia = **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric
* trošku vylepšená Binary Trie
  + eliminuje porovnávání bitů, které určuje kam jít ve stromě dále, v každém vnitřním uzlu
  + místo toho se určí další node, který je pro určení cesty podstatný - u každého nodu si držíme pozici bitu, který je pro určení podstatný
  + je tedy možné přeskočit několik porovnávání až do místa, kde má porovnávání význam



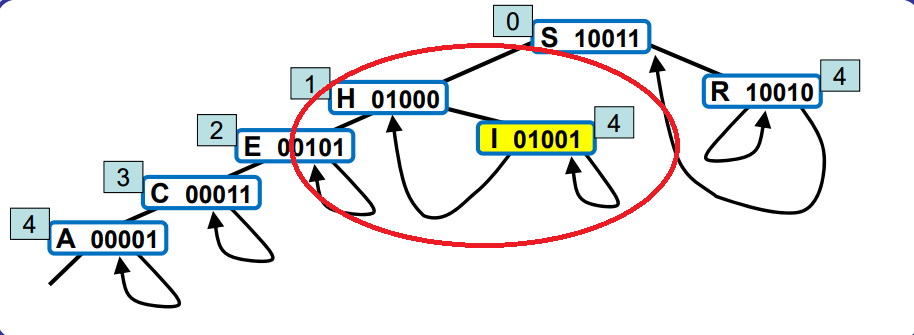
zde např. hledáme R (10011)

* začneme S, která říká v modrém čtverečku, že máme pořešit 0-tý bit - ten je 1 a proto jdeme doprava
* jsme v R a máme pořešit 4 bit, který je 0, takže jdeme doleva, což nás vezme smyčkou znovu do Rka. Porovnáme jestli je v nodu do kterého jsme došli opravdu R - ano je, takže jsme ho našli

teď např. hledáme B (00001)

* začneme v S - 0-bit je 0 jdeme doleva
* v H máme kontrolovat 1-ní bit - ten je 0 -> doleva
* v E máme kontrolovat 2-hý bit - ten je 0 -> doleva
* v C máme kontrolovat 3-tí bit - ten je 0 -> doleva
* v A máme kontrolovat 4-tý bit - ten je 1 -> doprava - to nás vezme zpět do A - tzn máme určený node ve kterém by mělo být hledané B, jenže není, neboť tam je A .. tzn. našli jsme pozici ale nesprávnou hodnotu.
* povšimněme, že v tomto případě jsme nic nepřeskakovali a kontrolovali v podstatě každý bit, takže jsme si oproti klasickému search v binary trie moc nepomohli

**Insert** **operace**

* najdeme správné místo opět podle algoritmu popsaného výše - koukáme jen na ty bity, na které máme a pokud nás to vezme do stejného nodu 2x, tak porovnáme ten node, kde jsme a ten který vkládáme, v jakém bitě se liší, tuto informaci u vkládaného nodu uložíme a node vložíme ,,do té smyčky”
* lepší příklad - chceme vložit do obrázku výše I = 01001
  + jsme v S - 0-tý bit 0 - doleva
  + jsme v H - 1-ní bit 1 - doprava
  + to by nás vzalo zase do H, tzn. že tady máme I vložit.
  + H 01000 a I 01001 se liší až ve 4-té bitě, tzn. ten bude rozhodovací - tuto informaci k I uložíme a I vložíme
  + výsledek vypadá takto
  + 

# 5. Vyhledávání v textu založené na konečných automatech.

Konečné automaty (jen ve zkratce co se řeší v ohledně tohoto v PALech)

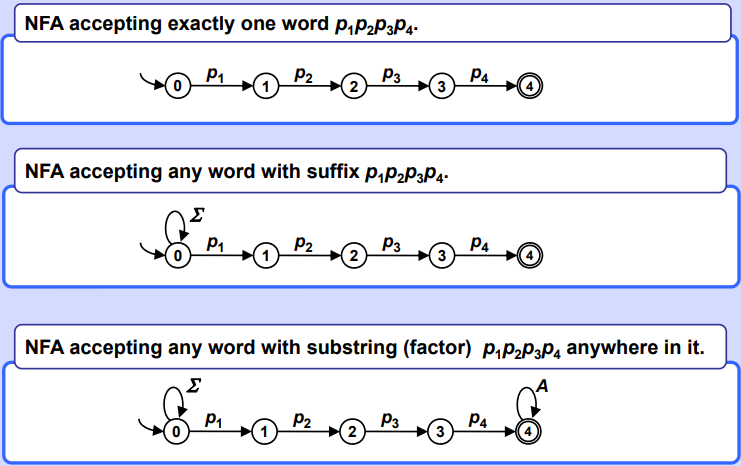
* mají nějaký startovní node a koncové nody, které přijímají slovo, abecedu, kterou lze použít pro přechody
* v PALech se řeší jak deterministické tak nedeterministické (i epsilon přechody)
  + nedeterministické nemusí mít u každého uzlu definované přechody pro každý symbol abecedy nebo naopak mohou stejným symbolem jít do více stavů
  + při přechodu u NFA přecházíme do množiny stavů, ne jen do jednoho
  + pokud bude v množině stavů na konci slova alespoň jeden stav konečný, slovo je automatem přijímáno
* NFA se převádí na DFA redukcí
  + začneme ve startovním a tvoříme novou tabulku, pokaždé, když přejdeme do nějakého stavu co ještě není v tabulce, tak přidáme řádek (když v NFA máme přechod do více stavů stejným písmenem, tak je mergneme do nového stavu, např. ze stavů 3, 4 uděláme stav 34)
  + výsledné DFA může být o dost víc komplexnější než nfa - může narůst až exponenciálně
  + nfa jsou jednodušší pro konstrukci

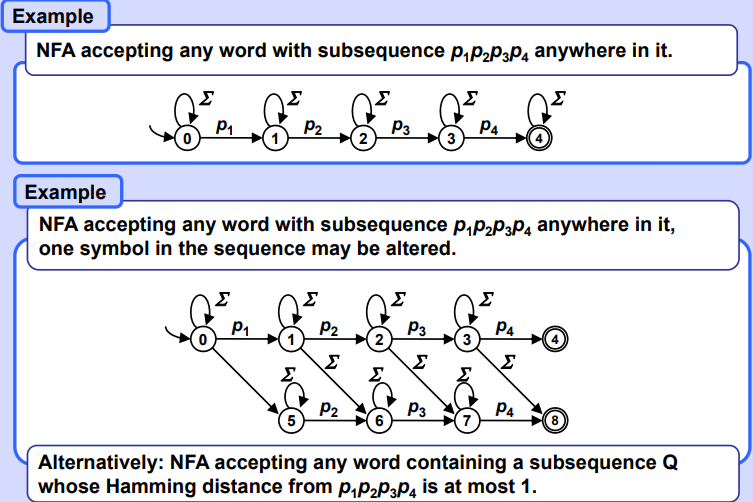
### 5.1 Text search

#### 5.1.1 Automaty a jazyky

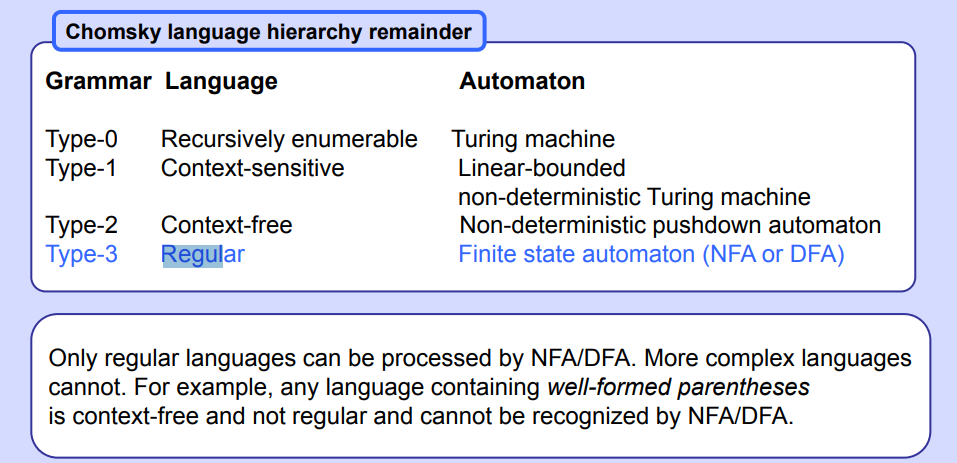
* hledáme nějaký pattern v textu
* k hledání v textu se právě hodí konečné automaty

**Příklady automatů**

****

****

**Chomského hierarchie jazyků**

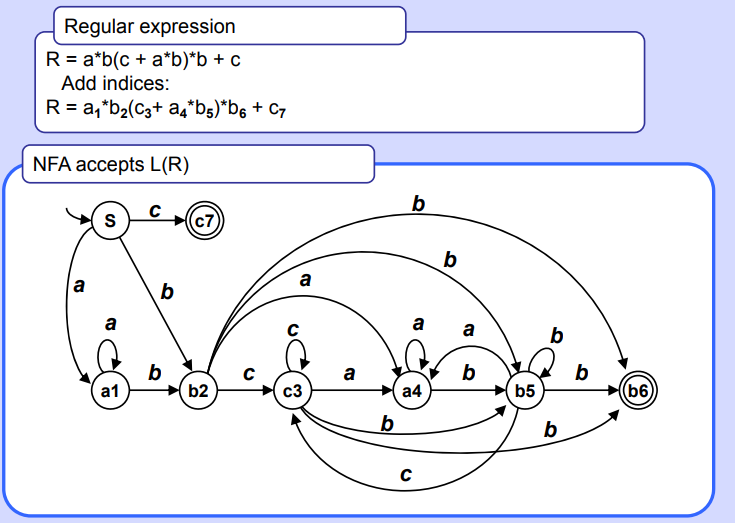


**Operace nad regulárními jazyky**

* Sjednocení jazyků = množina slov obsahující slova z L1 i L2
  + u automatů se provádí tak, že oba automaty nakreslíme vedle sebe, vytvoříme nový startovní uzel, ze kterého vedeme epsilon přechod do startovních uzlů obou automatů
* Zřetězení jazyků = množina slov, která jsou vytvořena tak, že dáme za sebe slovo z L1 a slovo L2
  + vedeme ze všech koncových stavů prvního automatu epsilon přechod do startovního stavu druhého automatu
* Kleeneho operátor na jazykem L- označuje se \* - zřetězení jakéhokoliv počtu slov (i 0) z jazyka L a to v jakémkoliv pořadí
  + vytvoříme nový počáteční stav (který bude zároveň koncový, neboť kleeneho operátor může řetězit i 0 slov) a vedeme epsilon přechod do původního startovního stavu
  + z koncových stavů vedeme epsilon přechod do původního počátečního stavu
* všechny tyto tři operace vyústí opět v regulární jazyk - regulární jazyky jsou na tyto operace tzv. uzavřené
  + další operace jsou **průnik, rozdíl jazyků, doplněk jazyku**

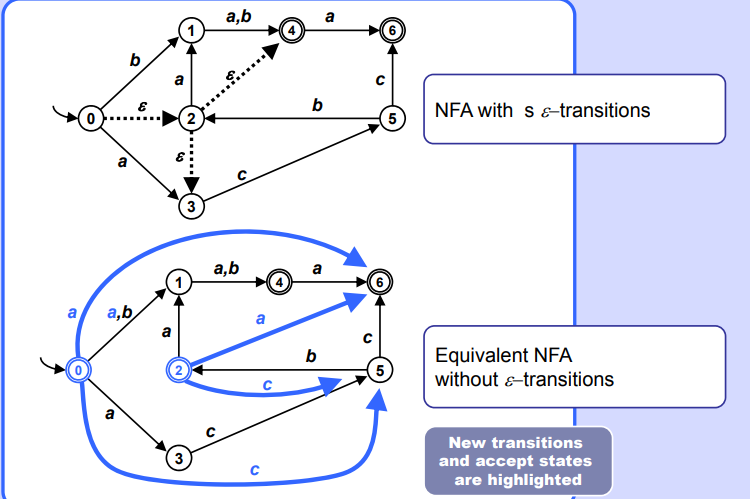
**Regulární výrazy**

* prázdné slovo je regulárním výrazem
* každý symbol z abecedy je regulárním výrazem
* kazdý string, který vznikl sjednocením, zřetězením nebo použitím kleeneho operátoru z výše zmíněného je také regulárním výrazem
* Reguární výraz lze převést na NFA



**NFA s epsilon přechody**

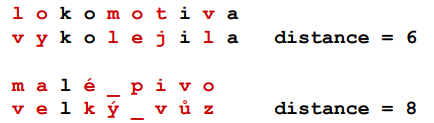
* epsilon přechod - přechod do jiného stavu bez jakéhokoliv symbolu abecedy
* **Epsilon uzávěr**
  + množina stavů, do kterých lze ze stavu dojít pouze epsilon přechody
* k odstranění epsilon přechodů a vytvoření normální NFA je potřeba
  + pro každý uzel definovat si jeho epsilon uzávěr
  + z uzlu povede přechod do jiného uzlu pokud byl tento přechod definován alespoň v jednom z uzlů, které jsou součástí jeho epsilon uzávěru
  + pokud v epsilon uzávěru pro uzel byl konečný stav, bude uzel konečným stavem



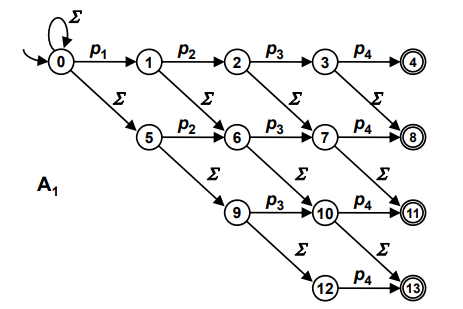
#### 5.1.2 Text search za pomocí konečných automatů

**Hammingova vzdálenost**

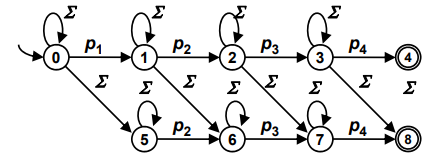
* k >= 0
* říká kolik písmenek bychom museli přepsat (nelze mazat ani vkládat) v jednom stringu, aby se z něj stal ten druhý, přičemž záleží na pozicích písmenek
* lze porovnávat jen u stringů stejné délky



Automat, který najde všechny výskyty substringů, které mají od patternu p1p2p3p4 Hammingovu vzdálenost nejvýše 3

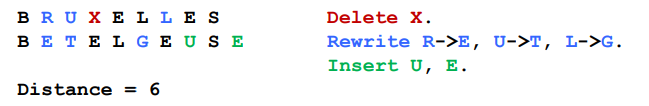


NFA jež akceptuje slova obsahující subsekvenci, jejíž hammingova vzdálenost od patternu p1p2p3p4 je **nejvýše 1**

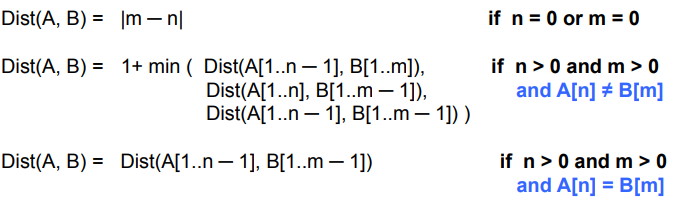


**Levenshteinova vzdálenost**

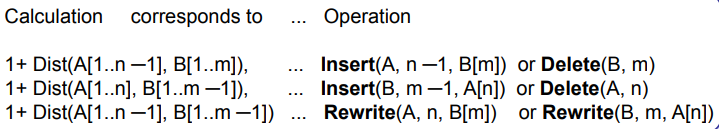
* oproti Hammingově vzdálenosti není nutné porovnávat stringy stejné délky, protože kromě přepisování umí i mazat či přidávat znaky
* je tedy definovaná počtem úprav, které musíme udělat, abychom z jednoho stringu dostali druhý
* úpravy mohou být různé, ale vzdálenost je vždy stejná a určená jednoznačně



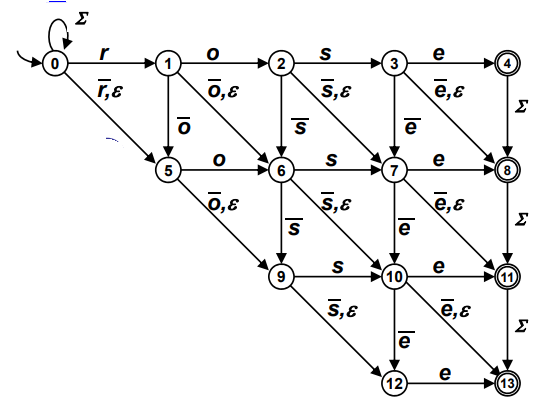
* pro výpočet se používá dynamické programování - definováno rekurzivně
  + nechť n je délka slova A (A má znaky a1.a2….an) a m je délka slova B (B má znaky b1.b2...bm)



Výpočtům odpovídají tyto operace

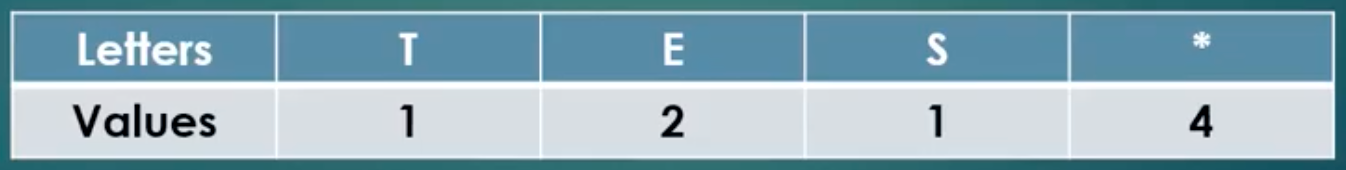


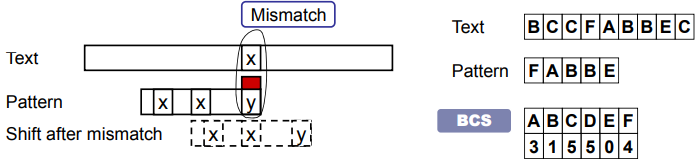
Automat, který hledá řetězec, jež má od patternu **rose** Levenshteinovu vzdálenost 3



epsilon přechody odpovídají operaci **Insert**, svislé přechody odpovídají operaci **Delete** a diagonální přechody operaci **Rewrite**

**Boyer-Moore**

* zarovnáme pattern se startem zadaného textu a porovnáváme znaky
* až dojdeme ke znaku, který se liší tak zkusíme shifnout pattern
* 3 případy
  + **Mismatch** se stal na posledním prvku v patternu - podle tzv. Bad Character Shift table (BCS)
  + ta říká pro každý znak o kolik shiftnou pattern doprava
  + kontrolujeme od zadu patternu - pokud tam není match, tak se podíváme co je v textu za znak a shiftneme pattern o tolik pozic doprava o kolik je uvedeno v této tabulce
  + neboli zarovnáme pattern tak, aby obpovídal prvnímu výskytu znaku **x** z textu v patternu
  + pokud se písmeno **x** v patternu nevyskytuje, tak se shiftneme o celou délku patternu
  + **BCSi = max(1, patternLenght - maxCharIndexInPattern - 1)**
    - Example (pattern = TEST)  
      



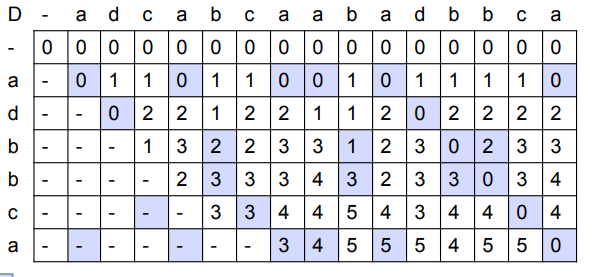
* **Mismatch se stal až po částečném matchi na konci patternu**
  + máme tedy nějakou příponu, která je správně

**Slovníkové automaty**

* automaty, které přijímají konečnou množinu slov - např. {“add”,”advanced”, “to”}

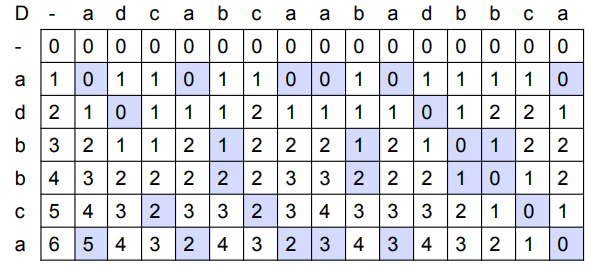
#### 5.1.3 Dynamické programování

**Hamming a DP**



* nahoře je text ve kterém hledáme
* vlevo je pattern, který hledáme
* první řádek je vždy vyplněn nulami
* tabulku vyplňujeme po řádcích následovně
  + pokud jsou písmena stejná sepíšu předchozí číslo z diagonály **tj. o jedno nahoru a doleva**
  + pokud jsou písmena jiná k předchozímu číslu z diagonály **přidám 1**
* **výsledkem jsou hammingovy vzdálenosti pro každé zarovnání patternu v zadaném textu**

**Levenshtein a DP**



* stejně jako u hamminga , nahoře je text ve kterém hledáme a vlevo je pattern
* první řádech jsou opět nuly a vyplňujeme po řádcích následovně
  + do políčka vložíme **minimum** z těchto hodnot:
    - **hodnota o 1 řádek výše + 1** - odpovídá smazání
    - **hodnota o 1 sloupec vlevo + 1** - odpovídá vložení znaku
    - **předchozí hodnota na diagonále** (o 1 nahoru a 1 vlevo) **+ 1**, pokud písmenka na aktuální pozici **nerovnají** (jinak + 0) - odpovídá přepisu či ponechání znaku
* výsledkem jsou opět levenshteinovy vzdálenosti pro každé zarovnání patternu v zadaném textu