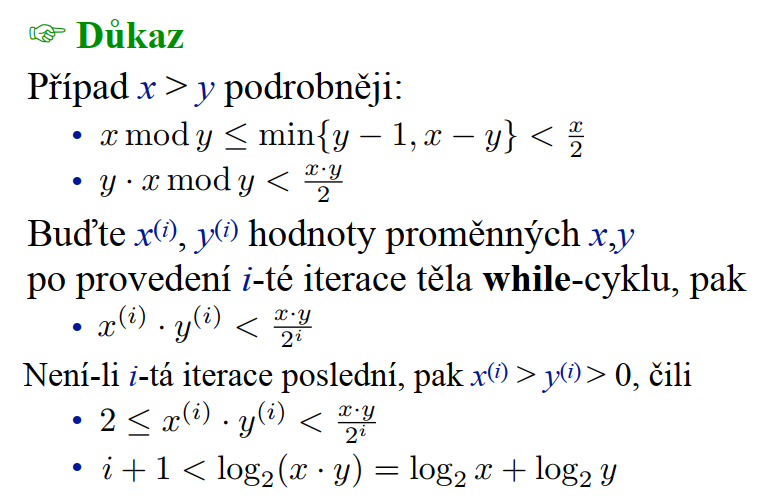
# Úvod

* Algoritmus - „Konečná posloupnost elementárních příkazů, jejichž provádění umožňuje pro každá přípustná vstupní data mechanickým způsobem získat v konečném počtu kroků příslušná výstupní data“
  + Konečnost
  + Hromadnost (obecnost)
  + Resultativnost (výstup)
  + Jednoznačnost
  + Determinismus
* Formální modely
  + Turingův stroj
  + RAM počítač
  + Rekurzivní funkce
  + Lambda kalkul
* Zápis algoritmů
  + Pseudokód
  + Python
* Ověření správnosti algoritmu
  + Konečnost
  + Částečná správnost
    - Pokud algoritmus na přípustnými daty skončí a obdržíme správný výsledek
  + Pokud je konečný a částečně správný → správný
* Invariant cyklu
  + Tvrzení, které je při každém průchodů cyklu pravdivé
* Porovnání efektivity algoritmů
  + Čas – počet kroků
  + Prostor – rozsah paměti
* Přístupy k analýze algoritmů
  + Nehorší případ
  + Nejlepší případ
  + Průměrný případ

# Teorie čísel

* Asymptotická notace
  + Funkce f(n) je třídy O(g(n)) právě když existuje c > 0 a n0 > 0 takové, že 0 <= f(n) <= c\*g(n) pro všechna n >= n0
  + Funkce f(n) je třídy Ω(g(n)) právě když existuje c > 0 a n0 > 0 takové, že 0 <= c\*g(n) <= f(n) pro všechna n >= n0
  + Funkce f(n) je třídy Θ(g(n)) právě když je třídy O(g(n)) a zároveň Ω(g(n)), lišící se pouze o multiplikativní konstantu
  + Funkce f(n) je třídy Õ(n) právě když existuje k takové, že f(n) je třídy O(g(n)\*logk(g(n)))
* Test prvočíselnosti
  + Délka vstupu je n = log2N + 1 (počet bitů)
  + Exponenciální složitost
  + Možná vylepšení
    - Stačí projít dělitele do
    - Omezení na lichá čísla
* Generování prvočísel
  + Eratosthenovo síto
  + Vylepšení
    - Prosívání od p2, namísto 2p
    - Vyloučení sudých čísel
  + Složitost O(N log log N)
* Největší společný dělitel
  + Euklidův algoritmus
    - d|x a d|y ↔ d|x-y a d|y pro x>y
    - Mohu postupně odečítat y
  + Jiná verze
    - d|x a d|y ↔ d|y a d|(x mod y)
    - x mod y = 0 ↔ y|xzW
  + Důkaz  
    
* Vyhodnocení polynomu
  + Pro polynom stupně n přímím výpočtem je složitost Θ(n2)
  + Hornerovo schéma Θ(n) operací
* Převody mezi číselnými soustavami
* Rychlé umocňování
  + Rozdělím exponent na mocniny 2
  + Příklad x15
  + Namísto x\*x\*…\*x rozložím jako x8 \* x4 \* x2 \* x = (((x\*x)\*x)\*x)\*(x)

# Třídění I

* BubbleSort
  + Porovnává dvojice a vyměňuje
* SelectionSort
  + Najde minimum a zařadí jen na konec setříděné části pole (nebo maximum)
  + Lepší než BubbleSort
  + Jen n-1 výměn
  + Jen O(n) zápisů do pole
* InsertionSort
  + Vezme prvek a zařadí jen do setříděné části pole
  + Lepší než SelectionSort
    - Méně porovnání
* Binární halda o n prvcích
  + Operace přidej
  + Operace odeberMinimum a oprav haldu
  + Obě operace v O(log n)
  + Vlastnosti
    - Výška – počet hladin {0, 1, …, h}
    - Na i-té hladině je maximálně 2i
* HeapSort
  + Z n prvků postavím haldu v čase O(n log n)
  + N-krát odeberu minimum O(n log n)
  + Třídění v čase O(n log n)

# Třídění II

* MergeSort
  + Dvě setříděná pole můžu třídit v lineárním čase
  + Pole dělím rekurzivně na poloviny, tz. že v i-té iteraci má jedno pole délku
  + Na rozdělení celého pole na jednotková pole potřebují
  + N-krát slévám, celková složitost je tedy O(n log n)
* Vnější třídění
  + Úkolem minimalizovat IO operace
* Rozhodovací strom
  + Rozhodovací strom pro n prvků má listy jako všechny jejich permutace, tj. n!
  + Délka cesty z kořene do listu odpovídá počtu porovnání
  + Počet listů je , a zároveň počet listů
  + Tedy
    - Odhadnu n! zespodu jako
    - Každý porovnávací třídící algoritmus provede v nejhorším případě Ω(n log n) porovnání
* Třídění počítáním
  + Třídím prvky 1 – k
  + Vytvořím si pole s rozsahem 1 – k, na indexy podle prvků ukládám četnost
  + Pokud nelze použít prvky, každému definuji klíč
* Radixové třídění
  + Postupně třídím podle číslic, postupuji odzadu

# Vyhledávání

* Lineární vyhledávání
  + Postupně procházím pole
  + Zarážka
    - Hledaný prvek přidám na konec pole → vždy nalezen
    - Na konci jej musí odstranit
* Vyhledávání v uspořádaném poli → Binární vyhledávání
  + Postupně půlím interval
  + Invariant cyklu – hledaná hodnota leží vždy mezi horní a dolní hranicí
  + Časová složitost Θ(log n)
* Algoritmus pro výpočet odmocniny půlením intervalu

# Datové struktury

* Datová struktura
  + Definuje ji funkčnost, ne implementace
* Binární strom
  + Kořenový vrchol
  + Koncové vrcholy jsou listy
  + Každý vrchol má maximálně dva syny
  + Každý vrchol, krom kořenového, má právě jednoho předka
* Binární halda
  + Je binární strom, který:
    - Má v každé hladině maximum možných vrcholů
    - Poslední hladina se zaplňuje zleva
    - Splňuje podmínku haldového uspořádání
  + Operace
    - Přidej – vložení nového prvku
    - OdberKoren – odebrání minimálního nebo maximálního prvku (kořen)
  + Počet prvků n
  + Vytvoření haldy v čase O(n log n)
  + Implementace v poli
* Konstrukce haldy v logaritmickém čase
  + Přidám prvek na spodek haldy
  + Porovnám s rodičem
  + Pokud je menší, zachovám pořadí (pro max-haldu)
  + Pokud je větší, prohodím
  + → počet kroků je maximálně výška hlady, tj. O(log n), v hladě je n prvků, tedy O(n log n) pro všechny
* Konstrukce haldy v lineárním čase
  + TODO

# Datové struktury II

* Zásobník (LIFO)
  + Např. mechanismus volání fcí, prohledávání do hloubky, aritmetické výrazy
* Fronta (FIFO)
  + Např. čekající procesy, prohledávání do šířky, počítačové simulace
  + Cyklické využívání paměti (mod kapacita)
* Prioritní fronta
  + Podobné jako fronta, jen prvky se předbíhají podle priority
  + Prvky stejné priority zachovávají pořadí
  + Implementace
    - Spojový seznam, pole
* Slovník
  + Klíč – hodnota

# Rekurze

* Rekurzivní volání funkce – funkce volá sama sebe
* Rekurzivní algoritmus – dělí problém na menší podproblémy
  + Lze většinou implementovat i iterativně
* Ukončení rekurze podmínkou
* Algoritmy
  + Euklidův algoritmus
  + Faktoriál
  + Fibbonaciho čísla
    - Pomocné pole hodnot pro již spočítané výslekdy
  + Rychlé umocňování matic

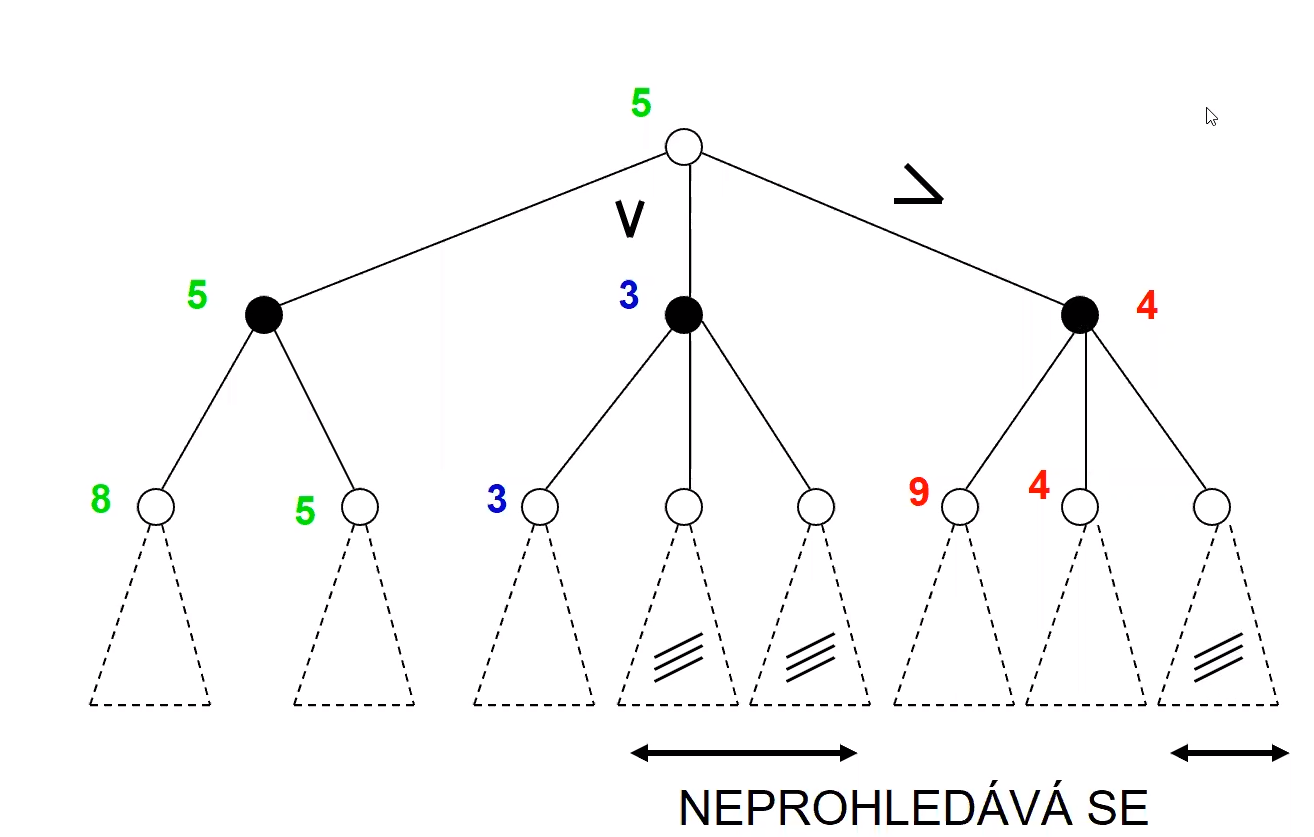
# Rekurzivní generování

* Algoritmy
  + Vypsat všechna k-ciferná čísla
  + Variace s opakováním
  + Kombinace bez opakování
  + Doplnění znamének
  + Rozklad čísla
* Binární strom
  + Průchod binárním stromem do hloubky
    - PREORDER – nejprve vrchol, pak postupně jeho podstromy
    - INORDER – nejprve levý podstrom, pak vrchol, pak pravý podstrom
    - POSORDER – nejprve oba podstromy, pak vrchol
    - Složitost O(n)
  + Průchod do šířky
* Binární vyhledávací strom
  + Ukládání dat podle klíče
  + Při vyhledávání stačí projít cestu od kořene k listu
* Aritmetický strom
  + Vyhodnocování aritmetických výrazů
* Obecný strom
  + Maximální počet synů každého vrcholu

# Grafy

* Vrcholy, počet N
* Hrany, počet M
  + Maximální počet u neorientovaného grafu,
* Rozlišujeme graf a jeho nakreslení
* Orientovaný a neorientovaný graf (obojí = obecný)
* Multigraf
* Posloupnosti vrcholů
  + Sled
  + Tah (sled bez opakujících se hran)
  + Cesta (tah bez opakujících se vrcholů)
  + Cyklus
* Souvislost grafu
  + U orientovaného
    - Silná souvislost → mezi každými dvěma vrcholy existuje orientovaná cesta
    - Slabá souvislost → obsahuje souvislý neorientovaný podgraf
* Strom (acyklický a souvislý)
  + Zakořeněný strom
    - Ostatním vrcholům přiřazujeme orientaci směrem od kořene
  + List
    - Vrchol stupně 1
* Les (pouze acyklický)
  + Jeho komponenty souvislosti jsou stromy
* Kostra grafu
  + Minimální kostra
* Bipartitní graf
  + Právě když neobsahuje cyklus liché délky
* Topologické uspořádání orientovaného grafu
  + Očíslovaní vrcholů tak, že pro každou hranu i → j platí
  + Orientovaný graf je acyklický právě když má topologické uspořádání
* Reprezentace grafů v programu
  + Matice sousednosti, matice vzdálenosti
    - Vhodné: Grafy s velkým počtem hran
    - Nevhodné: Grafy s velkým počtem vrcholů a malým počtem hran
  + Seznam následníků
    - Primárně pro orientované grafy
    - U každého vrcholu je uloženo, do kterých z něj vede hrana
    - Implementace
      * Matice N\*(N-1)
        + Z každého vrcholu může potenciálně vést hrana do n-1 vrcholů
      * V pythonu seznam seznamů
        + Dynamický, ale pomalejší
      * Matice N\*R
        + Pokud známe maximální počet hran, který vede z nebo do vrcholu
  + Seznam hran
    - U každé hrany uložena čísla koncových vrcholů
    - Vhodné: Algoritmy založené na zpracování hran
    - Nevhodné: Algoritmy vyžadující určení sousedů daného vrcholu, složitost O(M) … M je počet hran
  + Matice incidence
    - Matice velikost M\*N (hrany \* vrcholy)
    - Orientaci hran vyjadřujeme znaménkem
    - A[u][h] = +1 právě když hrana h vede z vrcholu u
    - A[u][h] = -1 právě když hrana h vede do vrcholu u
    - A[u][h] = 0 … jinak
    - Vhodné: Vrcholy s málo hranami, neefektivní
  + Dynamická reprezentace grafu
    - Vrchol je objekt se seznamem odkazů na sousední vrcholy
    - Pokud graf není souvislý, je třeba zvlášť seznam všech vrcholů
* Grafové problémy, 1. část
  + Určení souvislosti a komponent souvislosti
  + Nalezení cyklu
  + Nalezení kostry a minimální kostry
  + Určit bipartitnost
  + Řešení → prohledávání do hloubky nebo do šířky
* Grafové problémy, 2. část
  + Určení nejkratší vzdálenosti jednoho vrcholu od jiného
  + Určení nejkratší vzdálenosti mezi jedním a ostatními vrcholy
  + Řešení → prohledávání do šířky
* Prohledávání grafu do hloubky (DFS – depth-first search)
  + Začnu u vrcholu
  + Vyberu nejbližší, přesunu se k němu
  + Opět vyberu jeho nejbližší atd…
* Prohledávání grafu do šířky (BFS – breadth-first search)
  + Začnu vrcholu
  + Projdu všechny jeho sousedy
  + Přesunu se na nejbližšího souseda
* Řešení grafových problémů
  + Souvislost neorientovaného grafu
    - Spočítám všechny spojené vrcholy a zjistím, zda číslo odpovídá N
    - Složitost O()
  + Všechny komponenty souvislosti neorientovaného grafu
    - Opakovaně procházím graf z nějakého nenavštíveného vrcholu a počítám komponenty
    - Složitost O() nebo O(N + M)
  + Existence cyklu v neorientovaném grafu
    - Projdu všechny vrcholy a pokud se do některého dostanu podruhé našel jsem cyklus
  + Kostra souvislého neorientovaného grafu
    - Procházím hrany, když najdu nějakou vedoucí do dosud nezařazeného vrcholu, přidám ji
  + Bipartitnost grafu
    - Provedeme průchod celým grafem Navštívené vrcholy značíme střídavě čísly 1 a -1. Pokud narazíme na hranu spojující dva vrcholy se stejnou hodnotou graf není bipartitní.
  + Nejkratší vzdálenost v grafu (neohodnocený graf)
    - Průchod do šířky
    - Každému přiřadíme hodnotu o 1 větší než hodnotu vrcholu, ke kterého přicházíme
    - Složitost O() nebo O(N + M)
  + Vzdálenost dvou daných vrcholů v grafu
    - Stejné jako hledání nejkratších vzdáleností s tím, že alg. ukončíme pro průchodu cílovým vrcholem
  + Nejkratší cesta v grafu
    - Vlna

# Prohledávání stavového prostoru do hloubky

* Stavový prostor obvykle celý neznáme (nemáme jej v paměti)
* Prohledávání do hloubky (zpětné prohledávání, DFS, backtracking)
  + „Prohledávání bludiště“
* Problémy
  + Osm dam na šachovnici
  + Proskákání šachovnice koněm
* Paměťová složitost
  + Výška stromu představuje všechny možné cesty výpočtu
  + Kořen = výchozí situace
  + List = řešení nebo slepá ulička
* Časová složitost
  + Obvykle exponenciální
  + Někdy hledám pouze jedno řešení
  + Pokud hledám všechna, musím projít celý strom
* Urychlení prohledávání do hloubky
  + Ořezávání
    - Dojdeme-li k uzlu, který je „neperspektivní“, dál již nepokračujeme a odřízneme jej
    - Příklad ořezávání
      * Osm dam na šachovnici
        + Bez ořezávání: vyzkouším všechna potenciální řešení, tj. všechna postavení a na konci ověřit kolize
        + S ořezáváním: umístit vždy jednu dámu na řádek a ihned otestovat kolize
      * Magické čtverce
        + Každý řádek musí mít přesný součet
  + Heuristika
    - „Intuitivní nápověda“, jak rychleji dospět k řešení
    - Upřednostním některé cesty před jinými
    - O žádná řešení nás nepřipraví, jen některá upřednostní
  + Ideál – propojení
* Algoritmus minimax
  + Hra dvou hráčů s úplnou informací
  + Bílý a černý se pravidelně střídají na tahu
  + Cíl algoritmu: pro aktuální pozici zvolit nejvýhodnější tah
  + Stavový prostor – strom hry
    - Kořen – výchozí pozice
    - Počet synů – počet možných tahů
    - Liché hladiny – bílý
    - Sudé hladiny – černý
    - List – konec hry s výsledkem
  + Pro malé hry lze napočítat vítěznou strategii
  + Výhodné postavení pro bílého – plusové body
  + Výhodné postavení pro černého – mínusové body
  + Hodnoty uzlů se počítají z hodnot synů
    - Když je na tahu bílý, vybírá si maximum z hodnot synů
    - Když černý vybírá si minimum
  + Velké hry
    - Z aktuální pozice se napočítá jen část stromu do určité hloubky (např. 4)
    - V konečné hloubce se listy ohodnotí statistickou funkcí
  + Zrychlení – ořezáváním
    - Ztrátové – strom se počítá jen do určité hloubky
    - Bezztrátové – alfa-beta ořezávání
      * Vynechávám ty podstromy, pro něž se protihráč nerozhodne
        + Tj. ty, které výsledek neovlivní  
          
* Prohledávání do šířky (breadth-first search, BFS)
  + V každé vrstvě vyzkouším všechny možnosti
  + Nutné pamatovat si všechny stavy v jedné vrstvě stromu
  + Paměťové náročné
  + Výhodné, pokud existují řešení blízká kořenu
* Implementace prohledávání do hloubky – **ZÁSOBNÍK**
  + vlož do zásobníku výchozí stav
  + dokud není zásobník prázdný nebo není nalezeno řešení opakuj:
    - vyjmi ze zásobníku vrchní stav a zpracuj ho vlož do zásobníku jeho všechna možná pokračování (pouze dosud nenavštívené stavy!)
* Implementace prohledávání do šířky – **FROTNA**
  + vlož do fronty výchozí stav
  + dokud není fronta prázdná nebo není nalezeno řešení opakuj:
    - vyjmi z fronty první stav a zpracuj ho vlož do fronty jeho všechna možná pokračování (pouze dosud nenavštívené stavy!)
* Příklady
  + Proskákání celé šachovnice koněm
    - Prohledáváni do šířky je paměťově nereálné
    - Každé možné řešení má 63 kroků, tedy musel bych vybudovat celý strom o výšce 63
  + Nejkratší cesta koněm na šachovnici
    - Algoritmus vlny
    - Prohledávání do hloubky neefektivní
      * Vím, že na libovolné pole se dá doskákat v 6 skocích
  + Procházení grafu (například určení komponent souvislosti)
    - Graf musíme projít celý → nezáleží, zda prohledávám do hloubky nebo do šířky
    - Možnosti realizace:
      * Rekurzivní procedura, která prohledává nenavštívené vrcholy
      * Zásobník a prohledávání do hloubky
      * Fronta a prohledávání do šířky
      * Náhodný výběr nenavštíveného vrcholu ke zpracování

# Rozděl a panuj

* Problém se rozdělí na dva podproblémy stejného typu, ale menší velikosti, z jejichž řešení lze snadno získat řešení původního problému. Každý podproblém je buď už triviální a vyřešíme ho přímo, nebo k jeho řešení použijeme stejný rekurzivní postup.
* Implementace rekurzivní funkcí
* Podmínkou je nezávislost řešených podproblémů
  + Protipříklad: Fibbonaciho čísla – neefektivní počítat vícekrát
* Vyhodnocení aritmetického výrazu
  + Princip
    - Ve výrazu najdeme znaménko, které se vyhodnocuje až jako poslední (tj. lineární průchod)  
      (je to znaménko zcela mimo závorky, z nich to s nejnižší prioritou, z nich to nacházející se ve výrazu co nejvíce vpravo)
    - Výraz rozdělíme na dva podvýrazy, vlevo a vpravo od znaménka
    - Oba podvýrazy vyhodnotíme (tj. rekurzivně)
      * Pokud je výraz triviální, např. „6“, vracím se z rekurze
    - S výsledky podvýrazů provedu operaci daným znaménkem
  + Složitost
    - V nejhorším případě je „poslední znaménko“ až na konci, tedy výraz procházím N+(N-1)+(N-2) → O(N2)
    - V nejlepším případě je znaménko vždy uprostřed → O(N log N)
      * Obdobně jako mergesort
    - V průměrném případě složitost O(N log N)
* Hanojské věže
  + Složitost O(2N) dána charakterem problému
* MergeSort
  + Třídění sléváním
  + Rozdělím pole na dvě stejně velké části a ty setřídím
  + Setříděná pole slevám do jednoho
  + Časová složitost O(N log N)
  + Paměťová složitost O(N log N)
    - Lze prodloužit na O(N)
* Quicksort
  + V průměru nejrychlejší známý třídicí algoritmus
  + Princip:
    - V seznamu zvolíme jeden prvek – pivot
      * Nalevo od něj zařadíme menší prvky
      * Napravo od něj zařadíme větší prvky
  + Implementace  
    def quicksort(s):   
     if len(s) <= 1: return s   
     x = s[len(s) // 2] # pivot   
     vlevo = [ a for a in s if a < x ]  
     stred = [ a for a in s if a == x ]  
     vpravo = [ a for a in s if a > x ]   
     return quicksort(vlevo) + stred + quicksort(vpravo)
  + Lze třídit i na místě – složitější algoritmus
  + Časová složitost v nejhorším případě O(N2)
    - Tj. když je pivot moc velký nebo moc malý
  + Časová složitost v nejlepším a průměrném případě O(N log N)
  + Snížení pravděpodobnosti volby nevhodného pivotu
    - Jeden náhodně vybraný prvek z tříděného úseku
      * Určité riziko volby nevhodného prvku
      * Mohu ověřovat, zda je v poli alespoň ¼ prvků menších a ¼ prvků větších
    - Medián ze tří náhodně vybraných
    - Hledání mediánu → neefektivní
* QuickSelect
  + Nalezení K-tého nejmenšího prvku
  + Po rozdělení seznamu podle pivota pracujeme dál jen s tou částí, ve které je hledaný prvek
  + V nejhorším případě složitost O(N2)
  + V průměrném případě složitost O(N)
  + Pokud budu volit pivota chytře, přibližuji se k O(N)

# Reprezentace aritmetického výrazu

* Reprezentace v binárním stromu
  + Listy jsou operandy
  + Vyšší vrstvy vyjadřují hierarchii znamének
* Notace aritmetického výrazu
  + Prefix, Infix, Postifx
* Vyhodnocení aritmetického výrazu v postfixové notaci
  + číslo → vložit do zásobníku
  + znaménko → vyzvednout ze zásobníku horní dvě čísla provést s nimi operaci určenou znaménkem výsledek operace vložit do zásobníku
  + konec → na zásobníku je jediné číslo = hodnota výrazu
  + Časová složitost O(N)
* Vyhodnocení prefixového výrazu
  + Průchod odzadu jako u postifixu
  + Zásobník na ukládání znamének a číselných hodnot
  + Nebo rekurze
* Převod infix → postfix
  + číslo → zapsat přímo na výstup
  + levá závorka → vložit do zásobníku
  + pravá závorka → tuto závorku zrušit, ze zásobníku postupně přenést na výstup všechna znaménka až k nejbližší uložené levé závorce, pak tuto levou závorku ze zásobníku zrušit
  + znaménko → vložit do zásobníku, předtím ale ze zásobníku postupně přenést na výstup všechna znaménka vyšší nebo stejné priority, nejvýše však k první uložené levé závorce
  + konec → ze zásobníku přenést na výstup všechna uložená znaménka
* Vyhodnocení výrazu v infixové notaci
  + Převod výrazu z infixu do postfixu v čase O(N)
  + Vyhodnocení postfixové notace v čase O(N)
  + Celková složitost je tedy O(N)
  + Převod a vyhodnocování se může provádět buď postupně, ale i současně
    - Současně → využívám 2 zásobníky
* Postavení aritmetického binárního stromu ze zápisu výrazu
  + Stejné jako vyhodnocování výrazu – ale místo operace přidávám do stromu

# Binární vyhledávácí stromy

* Datová struktura pro ukládání a vyhledávání dat podle klíče
* Pro každý vrchol platí:
  + Všechny záznamy uložené v levém podstromu mají menší klíč
  + Všechny záznamy v pravém podstromu mají větší klíč
* Výška binárního stromu (délka nejdelší cesty z kořene do listu)
  + Minimum – H = log2N
  + Maximum – H = N
  + Výška v průměrném případě H = log N
* Přidávání, vyhledávání i mazání v čase O(log N)
* Mazání hodnoty v binárním stromu
  + Smazanou hodnotu lze nahradit nejmenší hodnotou z pravého podstromu nebo největší hodnotou z levého podstromu
* Hledání nejmenší hodnoty → jdu pořád do leva
* Možnost použití zarážky

# Vyvažování stromů

* Dokonale vyvážený binární strom
  + Oba jeho podstromy mají počet uzlů rozdílný maximálně o 1
  + Výška log N
  + Obtížné jej udržovat vyvážený při přidávání a odebírání
  + Stavba: rekurzivně (TODO)
* AVL-strom
  + Výška levého a pravého podstromu se liší nejvýše o 1
  + Slabší požadavek, ale AVL-strom je maximálně o 45% vyšší, tj. 1.45 log (N)
  + Snadno udržitelný jako AVL-strom
* Postup implementace:
  + Postavím si dokonale vyvážený binární strom
  + Pak jej udržuji pouze jako AVL-strom

TODO: rekurzivní metody procházení binárního stromu (inorder, postorder, preorder)