Algoritmizace

Algoritmus

Eukleidův algoritmus, dělitelnost čísel

Test prvočíselnosti, Erathosthenovo síto

Rozklad čísla na cifry

Aritmetika s vyšší přesností (dlouhá čísla)

Hornerovo schéma, poziční číselné soustavy

Vyhledávání v poli – algoritmy

Třídění čísel v poli – Heapsort, Mergesort, Quicksort

Složitost problému třídění

Přihrádkové třídění – Counting Sort, Bucket Sort, Radix Sort

Dynamické datové struktury, Abstraktní datové struktury

Reprezentace dat v paměti

Zásobník, fronta, slovník, halda

Spojový seznam

Asymptotická časová i prostorová složitost a důaz správnosti algoritmu (konečnost, jednoznačnost)

Rekurze

Binární strom – preorder, inorder, postorder

BVS (binární vyhledávácí strom)

Notace aritmetického výrazu – vyhodnocení, převody

Hešovací tabulka

Vyvážené stromy

Obecný strom

Prohledávání do hloubky (DFS), zrychlení backtrackingu → ořezávání, heuristika

Prohledávání do šířky (BFS)

Minimax, alfa-beta ořezávání

Rozděl a panuj

Merge sort – rekurzivně

Quick Sort

Quick Select

Lineární algoritmus

Převod infix → postfix

Grafy

Prohledávání grafu do hloubky

Prohledávání grafu do šířky

Faktorové množiny

Erathosthenovo síto + Test prvočíselnosti

Test prvočíselnosti

1. zkoušet dělit od 1 do N
2. zkoušet dělit od 1 do N/2
3. nejlepší → dělit od 1 do $\sqrt{N}$

↳ protože $\sqrt{N} \cdot \sqrt{N} = N$ a pokud by číslo N bylo složené $N = a \cdot b$, tak

$a < \sqrt{N}$ v $b < \sqrt{N}$

↳ $O(\sqrt{N})$ časová

↳ $O(1)$ prostorová

ukázka:

```
def je_prvocislo(n):
    if n % 2 == 0
        return n == 2
    d = 3
    if d * d <= n:
        if n % d == 0:
            return False
        d += 2
    return True
```

Erathosthenovo síto

↳ zjištění všech prvočísel v daném intervalu

↳ algoritmus:

↳ vezmu první nevyškrtnuté číslo → vyškrtám násobky

↳ např. 2 a vyškrtám násobky: 4, 6, 8, ...

↳ vezmu další nevyškrtnuté číslo → vyškrtám násobky

↳ např. 3 a vyškrtám násobky: 6, 9, 12, ...

... atd.

↳ u čísla i začít vyškrtávat od i^2, protože menší $2*i$, $3*i$, ... vyškrtly už menší čísla

↳ $O(N/2 + N/3 + N/5 + \dots) = O(N \log(\log N))$ časová

↳ $O(N)$ prostorová

ukázka:

```
def erathosthenovo_sito(n):
    sito = [False, False] + [True] * (n-1)
    prvocisla = []
    i = 2
    while i <= n:
        if sito[i]:
            prvocisla.append(i)
            j = i * i
            while j <= n:
                sito[j] = False
                j += i
        i += 1
    return prvocisla
```

Euklidův algoritmus (dělitelnost čísel)

Pokud je $X < Y$ → pak $NSD(X,Y) = NSD(X, Y-X)$

Pokud je $X > Y$ → pak $NSD(X,Y) = NSD(X-Y, Y)$

Pokud je $X = Y$ → pak $NSD(X,Y) = X = Y$

↳ Existují 3 verze
- odčítací (pomalá)
- modulo
- rekurzivní

ukázky

modulo

```
while y > 0:
    x, y = y, x % y
return x
```

rekurzivní

```
def nsd(x, y)
    if y == 0:
        return x
    return nsd(y, x % y)
```

Důkaz správnosti

konečnost

↳ v každém kroku se součet $X + Y$ sníží o 1 a sečty musí někdy skončit

Prostorová a časová složitost

prostorová: $O(1)$ - využívá jen 2 proměnné, iterativní

"$O(\log(\min(X,Y)))$ → u rekurzivní"

časová: $O(X+Y)$ - u odčítací verze

$O(\log(\min(X,Y)))$ → u ostatních, protože se hodnoty sníží vždy minimálně o polovinu → nejhorší Fibonnaciho čísla

Hornerovo schéma

polynom: $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$

⌜ přímý: x^n spočítáme, krát a_n
x^{n-1} spočítat, krát a_{n-1}
...

→ počítání mocnin x^n je druhé $\frac{n(n+1)}{2}$ násobení

$\mathcal{O}(n^2)$ časová ⌟

Hornerovo schéma

↳ rychlejší umocňování $x^{25} = x^{16} \cdot x^8 \cdot x^1$ ⌜ $\mathcal{O}(\log N)$ ⌟

↳ vynásobit x tolikrát kolikrát se jde

↳ $P(x) = (\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0$

↳ $\mathcal{O}(n)$ časová → n násobení
→ n sčítání

```
def horner(koeficienty, x):
    vysledek = 0
    for koef in koeficienty:
        vysledek = vysledek * x + koef
    return vysledek
```

Poziční číselná soustava = šestnáctková, desítková, dvojková

↳ x → základ soustavy

↳ a_i → proměnná

$423 \rightarrow 4 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 423$ (funkce int() používá hornerova schéma)

Rozklad čísla na cifry

```
while n>0:
    cifra = n % 10
    print(cifra)
    n = n // 10
```

→ vypíše v opačném pořadí

→ $O(\log N)$ časová → podle počtu cifer

→ $O(1)$ prostorová , $O(\log M)$

↳ pokud vypisuji ↳ pokud ukládám

Aritmetika s vyšší přesností (Dlouhá čísla)

↳ počítání s čísly, co se nevejdou do standardních typů

↳ do pole (seznam) cifer → uložit odzadu (odpředu, nebo po skupinách jsem a používat %100, %1000

↳ [,] jednotky desítky

↳ sčítání

↳ po cifrách + carry → nový carry : (A+B+carry) // 10

výsledek : (A+B+carry) % 10

$O(N)$ časová → každá cifra jednou

↳ násobení

↳ každé s každým → $O(N^2)$ časová a pak sečíst mezivýsledek

↳ dělení

↳ $O(N^2)$

Algoritmy na vyhledávání v poli

Sekvenční

↳ porovnávání s každým prvkem

↳ O(N) časová

```
def najdi(pole, x):
    for i in range(len(pole)):
        if pole[i] == x:
            return i
    return -1
```

Se zarážkou

↳ přidání prvku na konec, takže ho vždy najdu. Ten důvod je kvůli while cyklu, abych nekontrolovala $i < len(pole)$ a zároveň $x == pole[i]$

↳ O(n) časová

Binární vyhledávání

↳ musí být setříděné pole

↳ princip půlení intervalů:

1) doprostřed pole
2) hledaný prvek $> x$ → jen pravá polovina a naopak
3) opakovat dokud nenajdu prvek nebo zbyde prázdný interval

↳ O(logN)

```
def binarni_hledani(pole, x):
    left = 0
    right = len(pole) - 1

    while left <= right:
        mid = (left + right) // 2
        if pole[mid] == x:
            return mid
        elif pole[mid] > x:
            right = mid - 1
        else:
            left = mid + 1
    return -1
```

Řazení dat v poli – přímé metody $O(n^2)$

↳ v poli od nejmenšího po největší

Select Sort

↳ projdeme nesetříděný seznam, najdeme nejmenší číslo a vložíme na konec setříděného seznamu, pak jdu od druhého prvku hledat nejmenší

↳ procházíme $n, n-1, n-2, \ldots, 2$

porovnání bude $n-1, n-2, \ldots, 1$ → $\frac{n \cdot (n-1)}{2}$

Insert Sort

↳ beru prvek z nesetříděného jeden po druhém a vložím v setříděném prvku na správné místo

↳ $O(N^2)$ nejhorší

Bubble Sort

↳ porovnám sousední dvojice a pokud levý > pravý, tak prohodím

↳ $O(N^2)$

↳ modifikace: → přeskládáním (mění směr zprava, zleva)

→ zkracuje průchod o $n-1$ (protože probublá na konec)

→ zkracuje průchod od poslední výměny, protože začátek už bude setříděné

Třídění v poli - Merge Sort

1) porovnání dvojic prvků ← dřív rozdělí na jednoprvkové, které jsou z principu seřazené
2) slévání dvojic do čtveřic atd. → zvětšování na dvojnásobek
3) jakmile délka úseku dosáhne n, tak je konec

↳ je stabilní, tedy nemění pořadí

↳ potřebuje pomocné pole o velikosti N

↳ $O(N \log N)$ časová složitost

↳ $\log_2 N$ kroků výpočtu, kde se v každém vykoná práce $O(N)$ na sliti prvků

↳ ideální pro vnější třídění, tedy třídění souborů na disku, ne v paměti

↳ slévání 12 13 20 67
3 17 54 55 → 3 12 13 17 20 54 55 67

↳ posouvám ukazatele u porovnávaných prvků

Vnější třídění → používá Merge Sort, protože je potřeba sekvenční čtení

↳ nelze indexovat

↳ dvoufázové třídění

↳ rozdělovací → rozdělí stříďavě do souborů A a B

↳ slévací → čte z A a B, slévá do souboru C

↳ přímé slučování → hloupě bere čísla a slučuje

↳ přirozené slučování → hledá už seřazené úseky a ty slučuje

↳ čtyřfázové třídění

↳ 4 soubory: AB - vstupní, C, D - výstupní

↳ bere úseky z A a B a výsledky stříďavě zapisuje do C nebo D, tak vznikne na konci v každém úsek délky N a C+D bude výsledky → méně operací

↳ složitost $O(N \log(N))$ → ale operací s diskem, takže 100 000 x pomalejší než RAM

↳ optimalizace

1) předtřídění v paměti

↳ v paměti Quicksortem, Heapsortem, nižší spousta kroků slévání

2) vyšší stupeň slučování

↳ jednofázové třídění ale s K vstupními a K výstupními soubory.

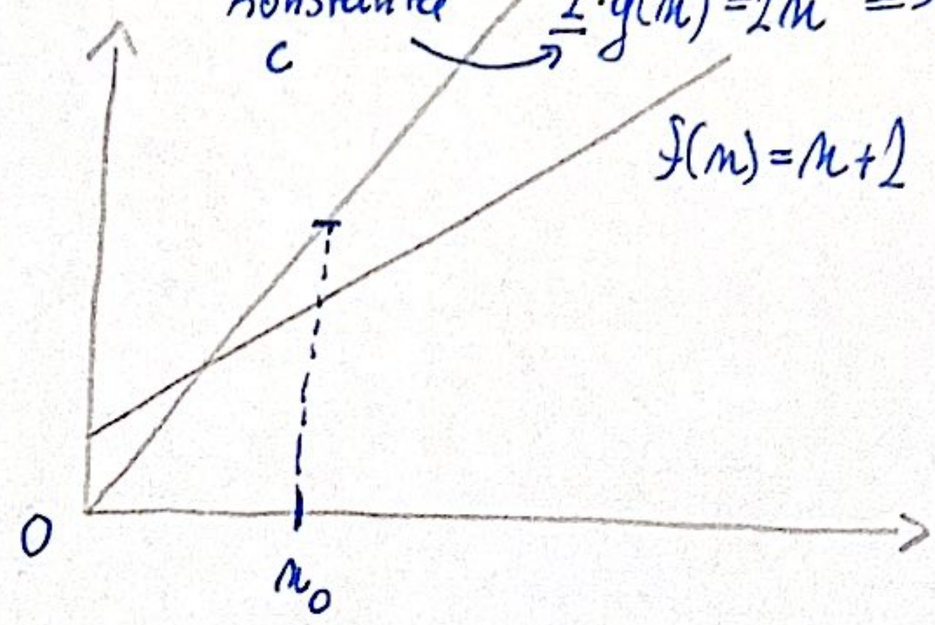
tedy časová složitost se sníží z $\log_2 N$ na $\log_K N$, ale $O(N \log N)$ je vlastně to samé

Asymptotická složitost algoritmu

$$f,g : \mathbb{N} \to \mathbb{R} : f(n) \in \mathcal{O}(g(n)) : \exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n > n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

odhad shora → velké $\mathcal{O}$

↳ např. platí
- $n+1 \in \mathcal{O}(n^2)$
- $n^2 \in \mathcal{O}(n^3)$
- $n+1 \in \mathcal{O}(n)$



$\mathcal{O}(g(n))$ → horní odhad

↳ $f(n)$ neroste rychleji než $g(n)$

↳ zkráceně $\exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n)$

↳ platí pro funkce $f,g : \mathbb{N} \to \mathbb{R} : f(n) \in \mathcal{O}(g(n))$

$\Omega(g(n))$ → dolní odhad

↳ $f(n)$ neroste pomaleji než $g(n)$ (tahle $f(n)$ roste rychleji než $g(n)$)

↳ zkráceně : $f,g : \mathbb{N} \to \mathbb{R} : f(n) \in \Omega(g(n)) : \exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n > n_0 : 0 \leq c \cdot g(n) \leq f(n)$

$\Theta(g(n))$ → těsný odhad

↳ platí pokud zároveň platí $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, takže platí $\mathcal{O}(g(n))$ i $\Omega(g(n))$, tedy algoritmus se chová přesně jak ta funkce

↳ počítané operace: aritmetické, porovnávání ... (časová)
- nepočítá velikost vstupních dat, jen další pomocná pole (prostorová)

Složitost problému

↳ s jakou složitostí se dá daný problém nejlépe vyřešit

↳ složitost algoritmu → složitost určitého, ne zrovna nejlepšího algoritmu

příklady:

↳ nalézt maximum z čísel $\Theta(n)$

↳ setřídit N čísel podle velikosti $\Theta(n \cdot \log(n))$

↳ např. heapsort, nebo lépe

Dolní odhad složitosti třídících porovnávacích algoritmů

↳ počet permutací → $N!$, algoritmus musí umět setřídit všechny možnosti, proto bude mít strom $N!$ listů

↳ h = výška stromu → maximální počet porovnání od kořene k listu

↳ s výškou h maximálně 2^h listů nebo-li $h \geq \log_2 L$, kde L je počet listů a tzn. $L \geq N!$

$$h \geq \log_2(N!)$$

$$N! \geq \left(\frac{N}{2}\right)^{\frac{N}{2}}$$

$$\log(N!) \approx \log(N^N) = N \log(N) \quad \Rightarrow \quad h \geq c \cdot N \log(N)$$

nebo-li $\Omega(N \cdot \log N)$ a $O(N \cdot \log N)$ $\Rightarrow$ $\Theta(N \log N)$

↳ složitost problému

Přihrádkové metody (s lineární složitostí)

↳ v $O(N)$, ale vysoká prostorová složitost

↳časová

Counting Sort → třídění počítáním

↳ pro třídění celých čísel s malým rozsahem

1) zjistit rozsah hodnot
2) vytvořit pole o velikosti rozsahu, naplněné nulami
3) projdeme data a hodnota určuje index v poli, kde inkrementuji o 1
4) projít pomocné pole a vypsat indexy, kolikrát je tam napsané

↳časová: $O(N)$

↳prostorová $O(M)$, kde M je velikost pomocného pole

Bucket Sort → přihrádkové třídění

1) dostanu seznamy a vytvořím si rozsah, podle kterého třídím, např. věk u dětí
2) projdu seznam a hodím hry prvky do pole kam patří
3) pak jen spojím přihrádky za sebe

- jiná implementace → pomocí čísel

↳ projdu prvky a inkrementuji tam, kde mají ten index, např. zase věk, prostě counting sort → do seznamu c

↳ jak pomocí (prefixu) změním ten seznam, aby tam byly hodnoty začínajících indexů místo počtu → zase v seznamu c

↳ projdu vstupní seznam a do výsledného seznamu např. b, který bude stejně velký jak vstupní, tak podle indexů v c umístím do seznamu b

↳ časová složitost → $O(N+R)$ → N → výsledný
→ R → seznam s indexy

↳ prostorová složitost → $O(N+R)$ → — " —

↳ je stabilní, zachová pořadí

↳ modifikace: čísla nejsou celá, ale desetinná při omezeném množství desetinných lze převést na celá

Radix Sort → více průchodové přihrádkové třídění

↳ řeší moc velká čísla

↳ budeme procházet a třídit do přihrádek pomocí cifer (nebo dvojcifer, nebo trojcifer) → nejdřív méně významné (jednotky, desítky), pak významnější

↳ díky stabilitě přihrádkového třídění bude výsledek správně

↳ např. pro čísla 1, 2, 1000 000 se pole zkrátí na 2x1000 místo 1000000 a pro každou trojici cifer musíme projít jednou → celkem 2 průchody

↳ $O(N \cdot \log R)$ → časová složitost, kde $\log R$ je počet průchodů a R velikost pole

↳ $O(N+R)$ → prostorová složitost

Reprezentace dat v paměti → nepodstatné i guess

↳ proměnná, deklarace

↳ statické, dynamické typování

↳ mutable → list, dict, set, uživatelské objekty

↳ immutable → int, float, bool, string, tuple

↳ seznam → realokace paměti

↳ pole → určení velikost při vytvoření

Operace se seznamem

↳ insert (index, hodnota) remove (hodnota) pop (index) del prvek → O(n) (časová složitost)

↳ append (hodnota) extend (seznam) pop() → O(1) amortizovaně ↗ kvůli realokaci

Dynamické datové struktury

↳ Lineární spojový seznam → ukazatel na další a hodnotu
→ prvky rozházené v paměti
→ rychlé vkládání/mazání, pokud se ví kam
→ nelze indexovat

↳ Binární strom → levý a pravý syn a hodnotu
→ hledání v O(logN) → pokud vyvážený
→ Binární vyhledávací strom a haldy

↳ spojová ne struktury

↳ klidně obousměrný spojový seznam → deque

Abstraktní datové typy

↳ definované svým chováním a nezáleží na implementaci ani na typu uložených dat

↳ zásobník, fronta, halda, slovník, vyhledávací strom

Zásobník

↳ LIFO → last in, first out

↳ odebírá se nejmladší prvek, pevné dno ← index 0 v poli, nebo konec LSS

↳ operace ← všechny mají konstantní časovou složitost O(1)

push → přidá prvek

pop → odebere prvek z vrchol

top → podívá se na hodnotu nahoře, ale neodebere

is_empty → prázdný nebo ne

↳ prohledávání do hloubky

↳ volání funkcí → např. vnořená funkce B v funkci A. A zavolá B. A uloží do stacku a pak z minulého slova nechá běžet A

Fronta

↳ FIFO → First in, First out

↳ operace

- append(x) → přidání prvek na konec O(1) (ENQUEUE)
- pop(0) → odebrání prvku O(n) → lineární složitost (DEQUEUE)

↳ řešení lineární složitosti odebrání:

a) ↳ posune občas, každé 10 odebrání → pořád lineární, ale průměrně lepší, potřeba pamatovat index

b) ↳ posune jen pokud přidání už nemá místo → konstantní odebrání amortizované konstantní přidání

c) ↳ cyklicky zacarovat nové prvky od začátku → obě operace konstantní

↳ v pythonu je deque → obousměrný lineární spojový seznam

↳ LSS → lehce odebírá ze začátku, těžce přidává na konci, takže s ukazateli

↳ Breadth-first search → prohledávání do šířky

Halda

↳ prvky musí být porovnatelné a neposunuje si pořadí příchodu prvků

↳ operace:

↳ přidat prvek → probublá se nahoru $O(\log N)$ (push)

↳ určí hodnotu minimálního prvku → $O(1)$, podívá se na vrchol

↳ odeber minimální prvek → $O(\log N)$, odebere se vrchol a vymění za poslední prvek a ten se propustí (pop)

↳ binární strom implementován v poli, zaplněný zleva zhora ← pro uložení do pole

↳ použití: - prioritní fronta → odebírání nejmenšího prvku
odec ≤ syn

- HeapSort → $O(N \log N)$, postavit haldu a pak odebírat minimum

↳ $H \approx \log_2 N$ → průměrný

↳ $H \geq N$ degenerovaný

↳ přidáním nového prvku na konec pole a porovná se s otcem a pokud je větší, tak se prohodí

↳ odebrání minimálního prvku → vezme se vrchol a vymění se za uzel v poslední hladině co nejvíce vpravo

→ porovnáme se syny a pokud větší, tak prohodíme s menším ze synů

↳ implementace v poli:

↳ a) kořen index 1, uzel s indexem i má syny na $2i$, $2i+1$

↳ listy uzel s indexem i má otce na $i//2$

b) kořen index 0, uzel s indexem i má syny na $2i+1$, $2i+2$ a otce na $(i-1)//2$

Heapsort (třídění haldou)

↳ postavit haldu → $O(N \log N)$ ← nebo lze postavit v lineárním čase $O(N)$

↳ rozebrat haldu → $O(N \log N)$ } celkově $O(N \log N)$

↳ třídí na místě → prostorová $O(1)$

↳ není stabilní → prohodí se pořadí prvků se stejnou hodnotou

Konstrukce haldy v lineárním čase $O(N)$

↳ bere se v poli jako binární strom, ale neuspořádaný

např.

↳ upravíme podstrom vlevo dole (už je)

↳ tuhle je „lepší" kořen

Prioritní fronta

↳ prvky se přeskládají podle priorit

↳ seznam (pole, LSS) → zařazujeme podle priority
→ nebo vybíráme podle priority

↳ samostatné seznamy pro každou prioritu

↳ halda → neuchovává pořadí se stejnou prioritou

↳ halda (s časem) → podle času se uchová pořadí

Slovník

↳ uchovává dvojice klíč - hodnota

↳ operace:

↳ vyhledej (klíč)

↳ vlož (klíč, hodnota)

↳ vymaž (klíč)

↳ implementace

↳ klíč → hešovací funkce → index v poli (z hešovací tabulky)

↳ u hešovací funkce mohou vznikat kolize

↳ minimalizace: → zvětšení hešovací tabulky

↳ řešení: → separační řetězení

↳ v hešovací tabulce bude seznam záznamů a ty projdeme a porovnáme s klíče → nejhůře lineární

→ otevřená adresace

↳ na hešovací tabulce pokud uložen záznam, tak půjde na index+1 atd., při odebrání prvku pořád plné, přehešování tabulky

přehešování při 1/3 zaplněnosti

řešení v Pythonu →

Binární vyhledávací strom

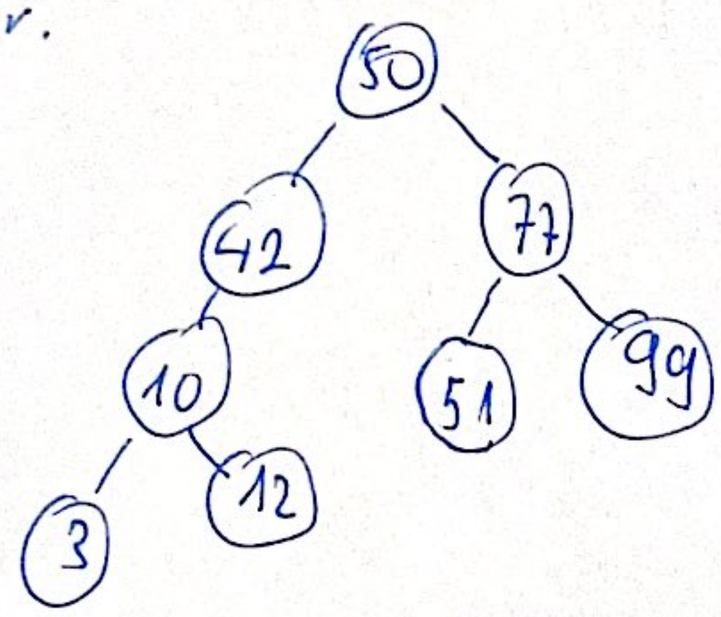
↳ datová struktura pro vyhledávání a ukládání dat podle klíče

↳ pro každý vrchol → v levém podstromu → menší

→ v pravém podstromu → větší

↳ musí platit pro všechny vrcholy a ne je pro bezprostřední syny

↳ např.



→ vyhledání

↳ v nejhorším: $O(N)$

↳ průměrném: $O(\log N)$

↗

stejné pro přidání hodnoty a odebrání

↳ mechanismus vyvážení → zachová výšku stromu $\log_2 N$

Spojový seznam

↳ uchovává hodnotu a ukazatel na další prvek (modifikace: i předchozí prvek)

↳ operace:

	LSS	pole
↳ přístup k i-tému prvku	$O(N)$	$O(1)$
↳ smazání prvku	$O(1)$	$O(N)$
↳ vložení na začátek	$O(1)$	$O(N)$
↳ vložení doprostřed	$O(1)$	$O(N)$

Rekurze

↳ je definován pomocí sebe sama

↳ rekurzivní algoritmus -> řešení menších instancí téhož problému

↳ rekurzivní volání funkce -> funkce volá sebe samu

↳ ukončení rekurze -> vzácně na podmínku

↳ příklady

↳ testování palindromu

↳ eukleidův algoritmus

↳ Fibonacciho posloupnost -> s memoizací

↳ nalezení rozmístění známek pro součet

↳ rozklad čísla na součet

Binární strom

↳ má haldu, binární vyhledávací strom a reprezentuje aritmetického výrazu

PREORDER → vypíše vrchol, pak jde do obou podstromů

INORDER → nejprve jde do levého, vypíše vrchol, pak do pravého podstromu

POSTORDER → nejprve jde do obou podstromů a vypíše (zpracuje) vrchol, až když odchází

průchod s rekurzí

```
projdi (x)
    if (x.levy != None):        <- preorder
        projdi (x.levy)
    if (x.pravy != None):       <- inorder
        projdi (x.pravy)
                                <- postorder
```

} print(x.info)

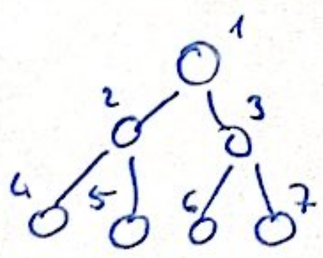
průchod bez rekurze

↳ do hloubky → zásobník

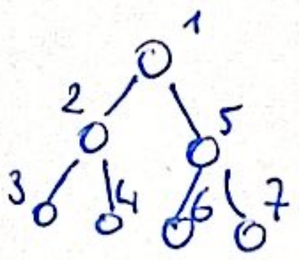
↳ do šířky → fronta

↳ to stejné, jen s frontou

```
do zásobníku (kořen)
dokud zásobník není prázdný:
    p = ze_zasobniku
    akce(p.info)
    if p.pravy != None: do zásobníku (p.pravy)
    if p.levy  != None: do zásobníku (p.levy)
```



zásobník:



↳ O(N) → každý vrchol navštíven právě jednou

Binární vyhledávací strom (znovu)

↳ vpravo vlevo menší, vpravo větší

↳ vyhledání rekurzivně: hledej (p, x):

```
if p == None
    return None
elif p.info == x:
    return p
elif x < p.info:
    return hledej(p.levy, x)
elif x > p.info
    return hledej(p.pravy, x)
```

→ $O(\log N)$

↳ přidání nového vrcholu → taky $O(\log N)$ → hledání až narazím na vrchol None a udržovat ukazatel o jednu zpět

↳ odebrání hodnoty → list = zruší se

→ 1 následník = posune se

→ více následníku = nesmaže se, ale nahradí se největší hodnotou z levého podstromu nebo nejmenší z pravého podstromu

→ taky $O(\log N)$

↳ hledání v BVS pomocí zarážky → odkazy z None nahradíme zarážkou

Vyvážené stromy → zajistit výšku $O(\log N)$

↳ počet uzlu v jeho levém a pravém podstromu se liší nejvýše o 1

Vyvážené stromy - pokr.

Postavení dokonale vyvážený binární strom

1. možnost

1. ukládané hodnoty vzestupně v poli

2. postavit vyvážený strom s n vrcholy bez info hodnot

```
def postav(n):
    if n==0:
        return None
    p=Vrchol()
    p.leny = postav((n-1)//2)
    p.praxy = postav(n-1-(n-1)//2)
    return p
```

3. pomocí in order dosadit hodnoty

2. možnost

ranou postavit z vzestupně seřazené posloupnosti a

```
def strom(a,x,y):
    if x>y:
        return None
    p=Vrchol(a[(x+y)//2])
    p.leny = strom(a, x, (x+y)//2 -1)
    p.praxy = strom(a,(x+y)//2+1, y)
    return p
```

AVL strom -> výškově vyvážený

↳ výška levého a pravého podstromu se liší nejvýše o 1

↳ počítá $O(\log N)$

↳ každý uzel má hodnotu vyvážení = výška levého podstromu - výška pravého podstromu

Přidání uzlu v AVL stromu

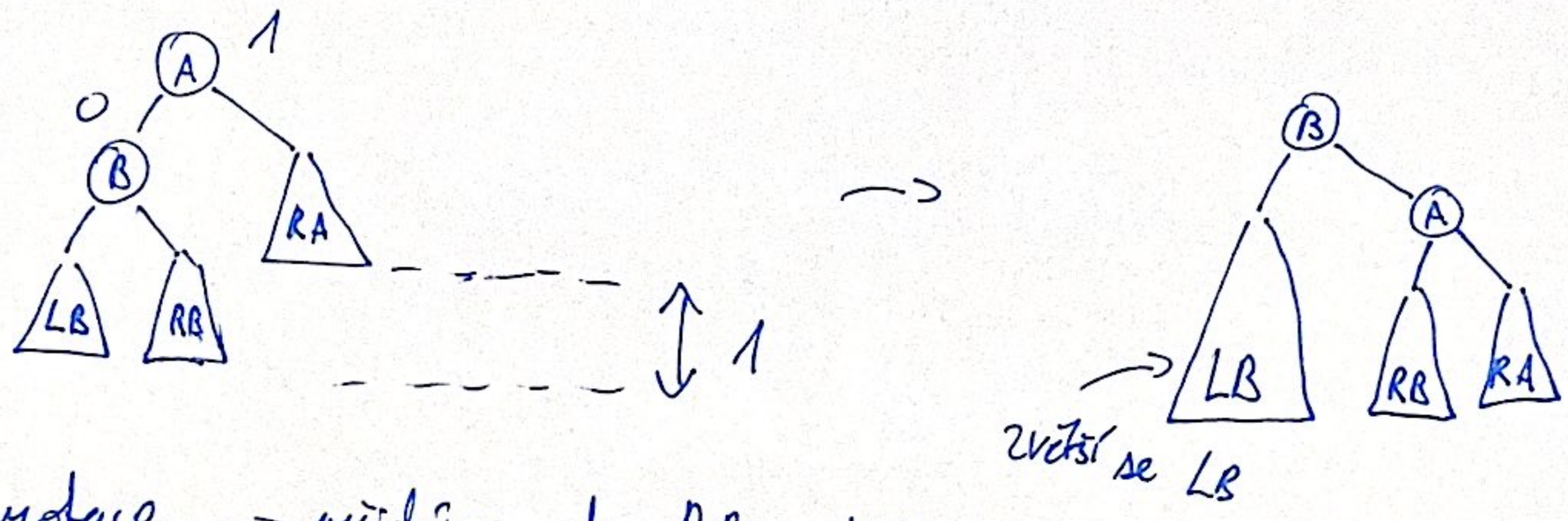
↳ vložím uzel na dno pomocí $O(\log N)$ jak v BVS stromu

↳ cestou si označím pivota, tedy nejbližší uzel uzlu, co bude mít balanci $\neq 0$

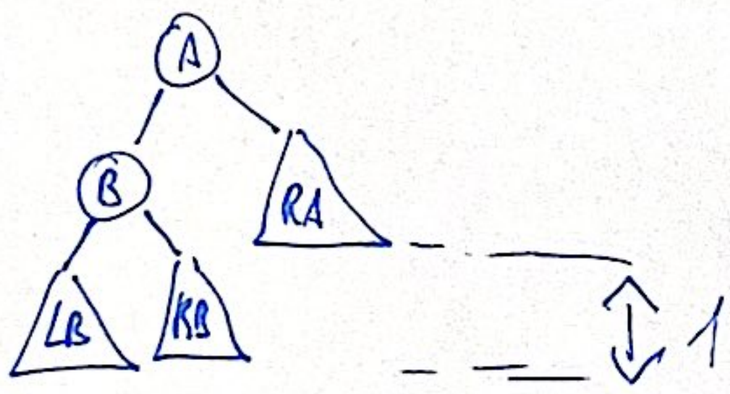
↳ postupuji pak směrem k pivotu a měním balanci uzlů

↳4 možnosti → pokud vložíme do vyššího podstromu pivota

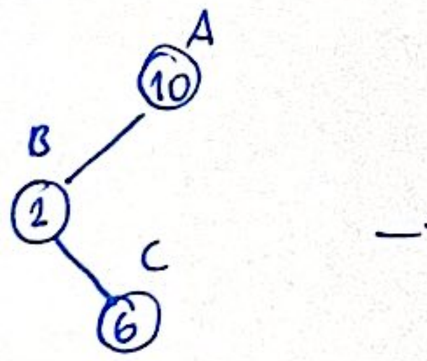
a) LL rotace → přidáme do LB



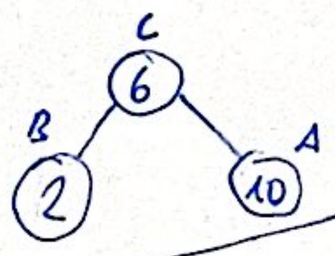
b) LR rotace → přidáme do RB, dělí se na 3 varianty



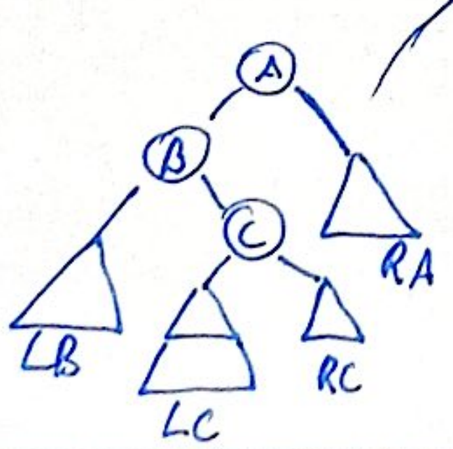
1) B je list → LB, RB i RA jsou prázdné



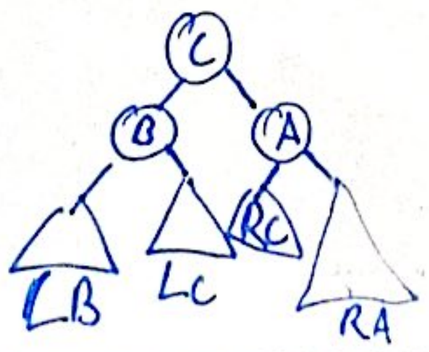
→



2) B není list → přidání do LC, tedy bude mít stejnou výšku jako LB a RA



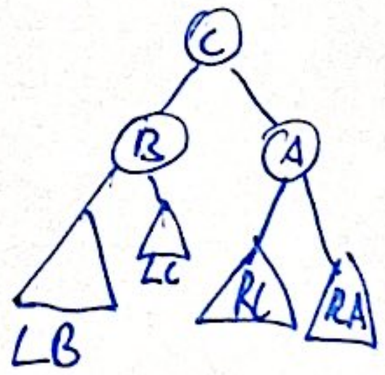
LR →



3) vložíme do RC

—11—

↳ LR rotace



Odebrání uzlu z AVL stromu

↳ 0 nebo 1 syn → vypustit uzel

↳ 2 syny → nahradit maximem z levého nebo minimem z pravého

↳ upravit na AVL

Obecný strom

1) známe maximální stupeň větvení → stejně jak u binárního

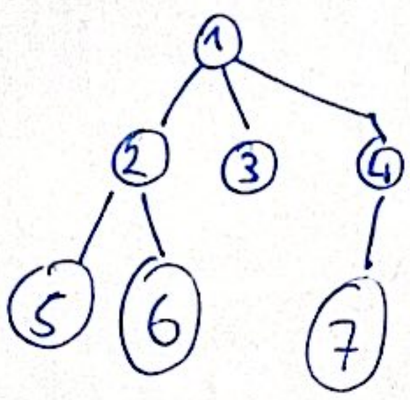
2) obecné → seznam s odkazy na syny

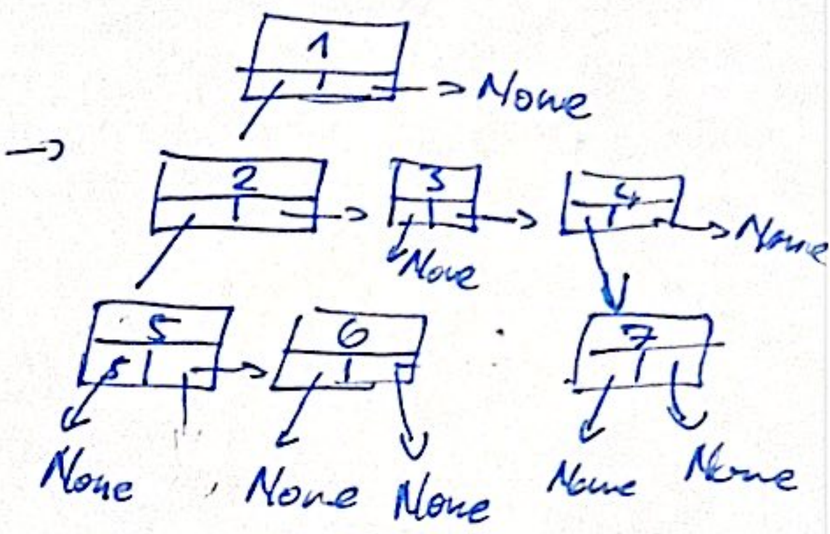
3) kanonická reprezentace

↳ pomocí binárního stromu

↓

syn a bratr





↳ písmenkový strom

↳ najdu slovo podle písmen → ukládání slov

↳ O(L), kde L je délka slova je časová složitost

DFS → prohledávání stavového prostoru do hloubky

↳ stavový prostor = množina všech možných stavů zkoumaného prostoru

↳ vrcholy = stavy hrany = přechody mezi stavy (orientovaný graf)

↳ generuje se během vyhledávání

↳ až najdeme řešení → musí být strom, aby se nezacyklilo

↳ do hloubky

↳ postupuje po jedné větvi dokud nedojde na konec, pokud slépá uličku, pak se vrací zpět → backtracking

↳ používá se - zásobník LIFO → nerekurzivní

- rekurzivní → např. s visited polem

↳ prostorová složitost → výška stromu

↳ časová složitost → počet uzlů

→ exponenciální k výšce stromu

→ řeší ořezáváním, heuristika

Ořezávání

↳ pokud podstrom nemůže vést k řešení → ořízneme

Osm dam (rozmístění bez kolizí)

↳ při umisťování každé dámy testovat kolize → 92

Magické čtverce (najít čtverce NxN, součty v řádcích i sloupcích (popř. diagonálách) stejné, čísla 1 až N^2 právě jednou)

↳ 816
↳ 357
↳ 492

↳ po vyplnění jednoho řádku otestovat

↳ součet čísel je od 1 ... do N^2 → $\frac{N^2 \cdot (N^2+1)}{2}$

↳ má mít podělit N řádů, tedy $\frac{N \cdot (N^2+1)}{2}$ je součet $N=3 \Rightarrow 15$

Heuristika

↳ odhad, kde je větší šance nalezení řešení → bude procházet v tom pořadí
nebo-li lišky se nakupí vlevo, kde je řešení

↳ nemusí zlepšit, ale nezhorší

↳ prohledáním šachovnice koněm → prozkoušet ty skoky, které vedou do pozic s menším počtem dalších pokračování

Warnsdorffův algoritmus

spojení s ořezáváním

↳ nejkratší cesta mezi dvěma městy → pamatujeme si v každém městě, jak nejlépe jsme tam přišli
→ průběžná délka větší než nalezená cesta → nemá smysl

Prohledávání stavového prostoru do šířky → BFS → algoritmus vlny

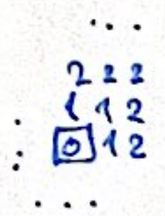
↳ v každé vrstvě stromu zkoušíme všechny cesty průběžně, než nenajdeme řešení

↳ pokud třeba najít nejkratší řešení (jinak do velké hloubky pomalé náročné, protože potřeba si pamatovat stavy)

implementace prohledávání do hloubky → zásobník a cyklus
implementace prohledávání do šířky → fronta a cyklus
} přidat jen nenavštívené

algoritmus vlny

↳ políčka označíme čísly podle toho, jak jsou vzdáleny od začátku



a pak zpátky: cíl → o 1 menší → ... → start

↳ nebo ukládat souřadnice ve frontě s souřadnicemi předchozího

průchod grafu → najít komponenty souvislosti

↳ označit vrcholy, pak BFS i DFS jsou stejně náročné
(kde jsem byl)

Minimax

↳ hra dvou hráčů s úplnou informací → bílý a černý se pravidelně střídají na tahu
→ vyhrát může jen jeden, nebo remíza

↳ stavový prostor = strom hry

↳ kořen → aktuální pozice

↳ počet synů → možné tahy

↳ liché hladiny → na tahu bílý

↳ sudé hladiny → na tahu černý

↳ list = konec hry (výhra nebo remíza)

↳ pro malé hry se prostoru celý strom hry (piškvorky, odebírání zápalek)

↳ hodnocení listu = +1 vyhrál bílý
-1 vyhrál černý
0 remíza

↳ z hodnocení listu se zdola určí algoritmem minimaxu hodnota kořene

↳ algoritmus minimaxu

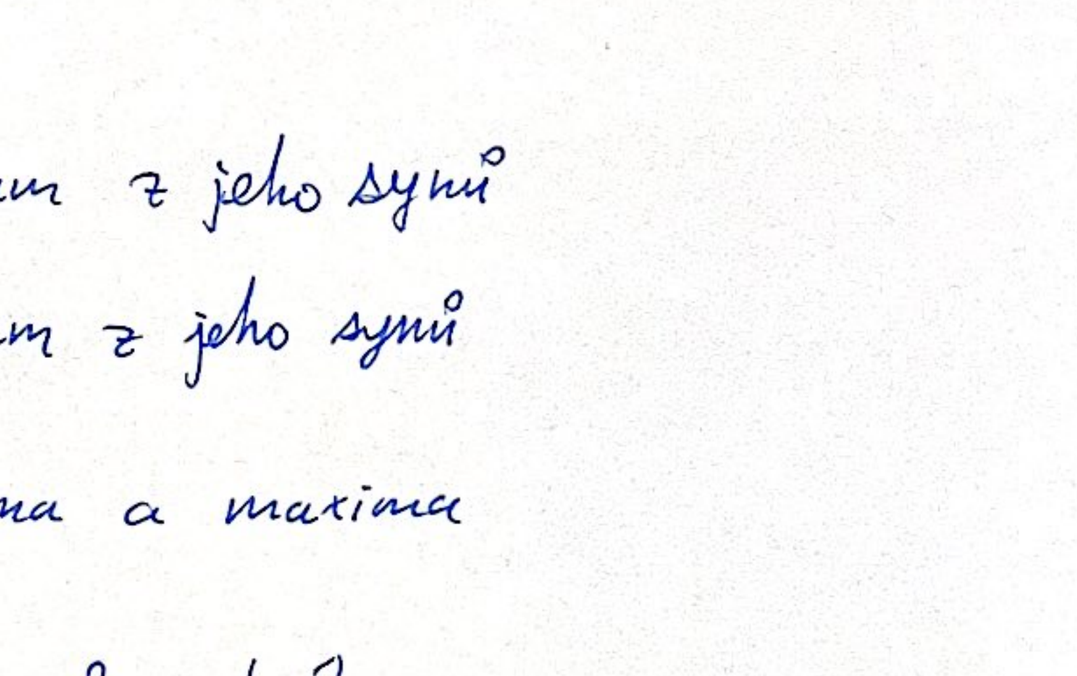
↳ bílý (lichá hladina) → vybírá maximum z jeho synů

↳ černý (sudá hladina) → vybírá minimum z jeho synů

↳ od listů se počítají střídavě minima a maxima

↳ pokud na tahu bílý a v kořeni → 1 = může vyhrát
0 = může remizovat
-1 = prohraje pokud černý neudělá chybu

↳ zvolený tah: v kořeni stejné číslo, co v dalším uzlu 1 max



realizace minimax (u velkých her)

↳ realizuje se prohledáváním do hloubky K, kde se ohodnotí → kladné číslo
↳ výhoda bílý
→ záporné číslo
↳ výhoda černý

↳ je potřeba vědět, který hráč je na tahu

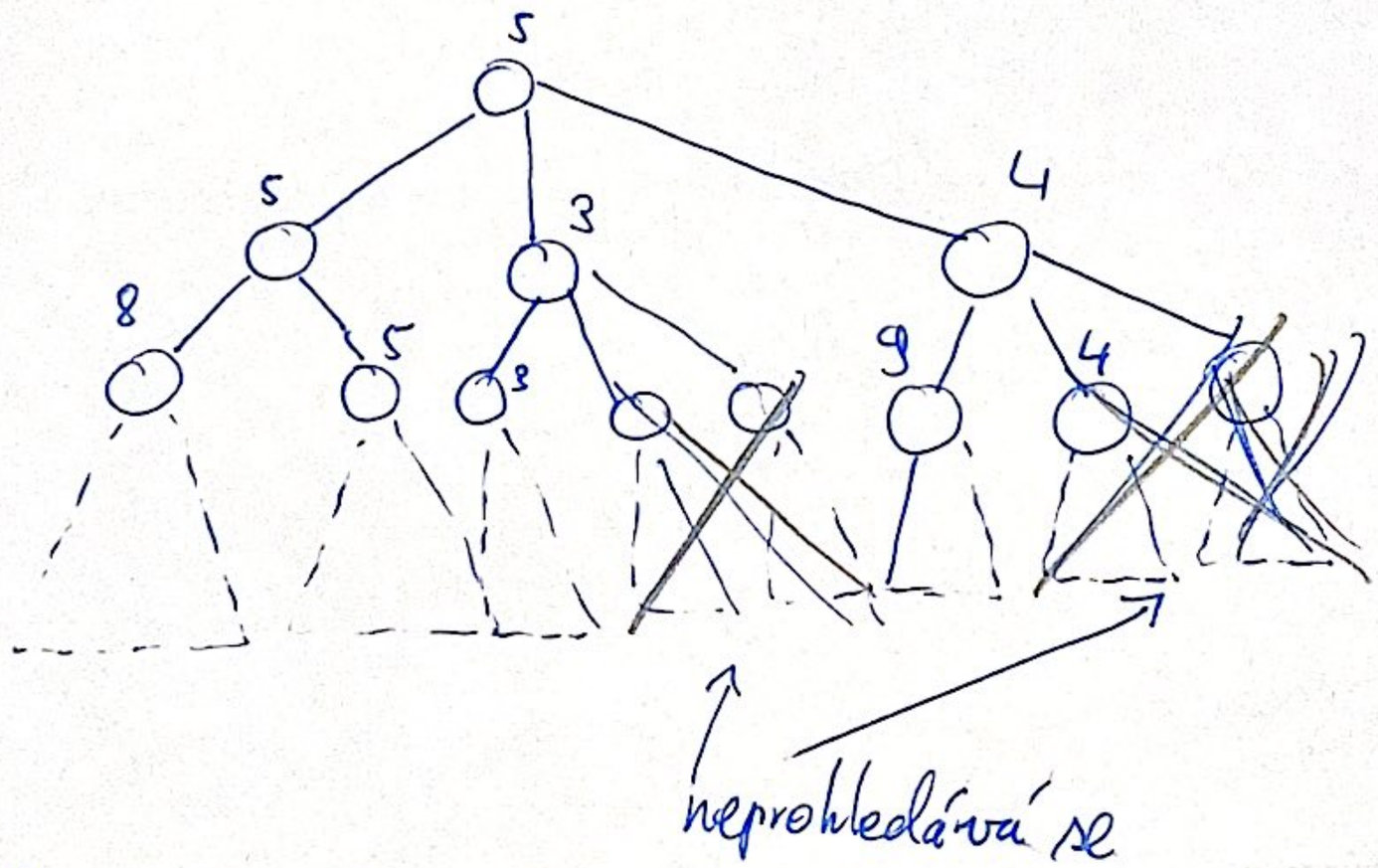
↳ negamax : $\min(a,b) = -\max(-a,-b)$ → aby se nemusely psát 2 funkce

ořezávání

↳ ztrátové: po málo vrstvách už neprovádělo špatně ohodnocené pozice

↳ kaskádové vyhodnocování

↳ bezeztrátové: alfa-beta prořezávání



↳ nemusím další větve prohledávat, když z mých z jedné větve jsem vyšel na 5 a u jiné větve by mě srazil soupeř na 3 nebo méně

Rozděl a panuj

↳ návrh rekurzivního algoritmu

↳ problém rozdělíme na 2 podproblémy menší stejného typu a pomocí jejich řešení lze získat jednoduše původní

↳ pomocí rekurzivní funkce

↳ musí být výpočty na sobě nezávislé, např. Fibonacciho čísla nebyla, protože sám se memoizovalo

Vyhodnocení aritmetického výrazu

↳ např. 5 * (3 + 4 * 6 - 8)

↳ 1) najít znaménko, co se vyhodnotí jako poslední
2) rozdělit na 2 podvýrazy → levo a vpravo od znaménka
3) vyhodnotit oba podvýrazy
4) provést ta poslední operaci

5 * (3 + 4 * 6 - 8) → $O(N^2)$ → výraz dělí na začátku nebo konci

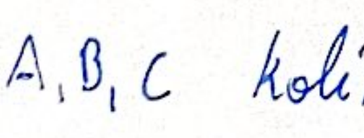
5 * (3 + 4 * 6 - 8) → $O(N \log N)$ → výraz dělí uprostřed

3 + 4 * 6 - 8

3 + 4 * 6

4 * 6

Hanojské věže



jediné řešení: 1) přenést N-1 disků z A na C (přenést N disků z A na B)
2) přenést disk z A na B
3) přenést N-1 disků z C na B

↳ $O(2^N)$ časová

Mergesort (rekurzivně pomocí rozděl a panuj)

1) rozdělit seznam na polovinu
2) setřídit levou i pravou část rekurzivně
3) slít oba seznamy do jednoho

```
def mergesort(s):
    if len(s) <= 1: return s
    mid = len(s) // 2
    return merge(mergesort(s[:mid]),
                 mergesort(s[mid:]))
```

$O(N \log N)$ časová

$O(N)$ paměťová → pomocné pole pro slévání

Quick Sort (třídění rozdělováním)

↳ nejrychlejší třídící algoritmus (v průměru)

```
def quicksort(s):
    if len(s) <= 1: return s
    x = s[len(s) // 2]
    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)
```

1) zvolím pivot a seřadím vlevo → menší
   vpravo → větší
2) rekurzivně setřídím
3) spojím obě části

$O(N \log N)$ → pokud bude volit vždy největší/nejmenší → $O(N^2)$

- vždy náhodně
- medián ze tří

$O(N)$ → paměťová

↳ pro rekurzi

$O(\log N)$

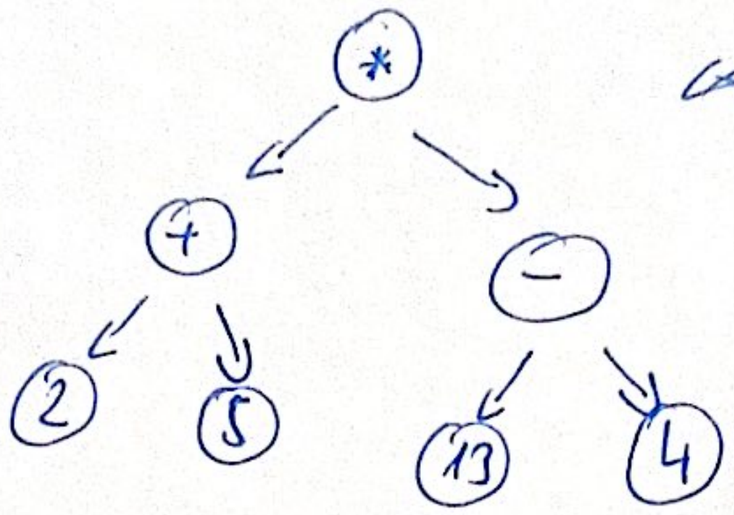
↳ při lepší implementaci

⌜ bez rekurze

↳ zásobník s dílky nebo-li částmi, které je třeba ještě setřídit ↵

Binární strom reprezentující aritmetický výraz

(2+5) * (13-4)



← postavení: jako vyhodnocení, ale vkládáme uzly stromů místo provádění operací

↳ listy stromu = čísla

↳ pořadí vyhodnocení je určeno strukturou stromu

Prefix → průchod v preorderu * + 2 5 - 13 4

Infix → průchod v inorderu 2 + 5 * 13 - 4 → ztratila informace o závorkách

Postfix → průchod v postorderu 2 5 + 13 4 - *

postfixový vyhodnocení O(N)

1) čísla do zásobníku

2) znaménko = vzít dvě čísla a provést operaci a vložit výsledek do zásobníku

↳ u nekomutativních → nahoře první operand, dole levý operand (operand = číslo)

Prefixové vyhodnocení

a) odzadu → to samé co postfix, ale změní pořadí operandů O(N)

b) 1) zásobník uložím znaménko
2) uložím číslo a pokud pod ním znaménko → nic O(N)
— II — číslo → provedu operaci

3) rekurzivní → číslice → vrátí hodnotu
znaménko → posune globální index a provede rekurzi O(N)

z infixu do postfixu → O(N)

↳ číslo → vypsat
↳ levá závorka → do zásobníku
↳ pravá závorka → vypsat všechna znaménka až k levé závorce uložené
↳ konec → ze zásobníku na výstup
↳ znaménko → do zásobníku, ale přenést všechny znaménka vyšší priority na výstup až do levé zá

Nalezení K-tého nejmenšího prvku → QuickSelect, Lineární Algoritmus

1) setřídit čísla v poli → S[k-1] $O(N \log N)$

2) postavit z čísel v poli haldu a z ní k-krát odebrat minimum

$O(N + k \cdot \log N)$

↑ postavení, R odebrání

3) postavit haldu z N-K+2 prvků

↳ holdu, aby k-tý prvek byl v haldě, kterého potřebujeme

↳ opakujeme k-2 krát:

↳ odebrat nejmenší
↳ přidat další

} práce s menší haldou

4) Quick Select

↳ po rozdělení podle pivota pracujeme jen s tou částí, kde je k-tý nejmenší prvek podle velikosti rozdělených seznamů

↳ v průměrném případě $O(N)$, v nejhorším $O(N \log N)$

↳ provede se quicksort

5) Lineární algoritmus

↳ podobný quickselectu, ale zvolit pivot chytře

↳ zvolí jako medián z mediánů pětic

Grafy

graf neorientovaný

graf neohodnocený

multigraf → 2 hrany spojují stejnou dvojici vrcholů
smyčka → počáteční a koncový vrchol hrany je shodný
} bez smyček a multihran

počet hran → 0 min

→ $\frac{N(N-1)}{2}$ max

→ $N \cdot (N-1)$ max u orientovaného

souvislost

- slabě souvislý = symetrizace je souvislým grafem
  - slabá komponenta souvislosti
- silně souvislý = každé 2 vrcholy jsou oboustranně dosažitelné
  - silná komponenta souvislosti

↗ acyklický orientovaný graf
silné komponenty souvislosti jednoduše

$|V| = |E| + 1$ → u stromu

v programu

- matice sousednosti → neorientovaný graf = symetrická
  - $O(1)$ zjištění existence hrany, ale $O(N^2)$ paměť, takže pokud málo hran a hodně vrcholů, tak špatné
- seznamy následníků
  - u každého vrcholu seznam, kam z něj vede hrana, hlavně pro orientované
  - a) seznam seznamů
  - b) max R hran → pak matice N×R
  - c) solidarity → velké pole kde jsou následníci + pole, kde je napsané, kde to pro daný vrchol začíná

- seznam hran
  - hrany v poli → uložen počáteční a koncový vrchol

- matice incidence MXN (hrany x vrcholy)
- dynamická reprezentace
  - například třída Vrchol obsahuje seznam odkazů na další vrcholy
  - neorientovaný → přidá hranu do obou vrcholů

Grafové problémy

1) souvislost
2) všechny komponenty souvislosti
3) obsahuje cyklus
4) určení kostry
5) zda je bipartitní

} lze řešit procházení grafu do hloubky nebo do šířky → protože mám pole visited, takže nemusíme vícekrát

6) vzdálenost dvou vrcholů
7) nejkratší cesta

} procházení do šířky

↑ + zpětný chod

Prohledávání grafu do hloubky

- označování navštívené vrcholy
- rekurze nebo zásobník + cyklus

Prohledávání grafu do šířky

- do šířky

} $O(N+M)$ → seznam následníků (N vrcholy, M následníci)
$O(N^2)$ → matice sousednosti

problémy + řešení:

souvislost → projdeme a inkrementujeme globální počítadlo o 1 a pak zkontrolujeme zda se rovná N (počtu vrcholů)

počet komponent → to stejné jen pokaždé začít u nenavštíveného vrcholu a inkrementovat jiné počítadlo

cyklus → pokud při průchodu máme cestu do již navštíveného vrcholu → konec, má cyklus, nepletl prv hrany, po kterých přišli

kostra → přidáme jen hrany, co vedou do nenavštívených vrcholů

bipartitnost → vrcholy označujeme 1 a -1 a pokud hrana spojuje 2 vrcholy se stejným označením - není bipartit.

Pokr. grafy

6. Vzdálenosti v grafu → do šířky, kořenému vrchol přiřadit o 1 větší

→ $O(N^2)$ $O(N+M)$ podle reprezentace grafu

↳ vzdálenost 2 vrcholů A a B → může se předčasně ukončit

7. nejkratší cesta v grafu → 2 fáze: a) algoritmus vlny

b) zpětný chod

Faktorové množiny

↳ jiný postup řešení problémů 1, 2, 3, 4 – kostra
– cyklus
souvislost komponenty

↳ „spojování množin" ne BFS, nebo DFS

DFU disjoint-find-union

↳ seznam hran (reprezentace)

↳ na začátku → graf bez hran

↳ každý vrchol jedna komponenta

↳ určení čísla (id) faktorové množiny → $O(1)$

↳ sjednocení faktorových množin pro vrcholy v a u → $O(N)$ počet hran, sjednocení

$O(M+N^2)$

1.+2. → počet komponent (faktorových množin) → sjednocujeme a snižíme o 1 pokud je komponent méně než kolik na 1 → souvislý a počet komponent je 1

3. → to samé, jen pokud hrana spojuje 2 vrcholy ve stejné faktorové množině tak cyklus

4. → to samé, ale hranu zařadíme do kostry jen tehdy pokud spojuje, že spojuje vrcholy z jiných faktorových množin