Nalezení *K*-tého nejmenšího prvku

z daných *N* čísel ($1 \leq K \leq N$)

speciální případ: $K = N/2$ … ***medián***

1. Setřídit čísla v poli S

výsledek: S[*K*-1] $O(N.\log N)$

2. Postavit z čísel v poli S haldu $O(N)$

z ní *K*-krát odebrat minimum $O(K.\log N)$

celkem časová složitost $O(N + K.\log N)$

3. Postavit v poli haldu z prvních *N*-*K*+2 prvků
K-2 prvků nechat stranou

Pak opakujeme (*K*-2)-krát:
- odebrat minimum z haldy (není *K*-tý nejmenší, je menší!)
- zařadit místo něj jeden prvek do haldy

Nakonec z haldy odebereme dvakrát minimum
= prvky v pořadí (*K*-1)-tý a *K*-tý nejmenší.

Časová a paměťová úspora – pracujeme s menší haldou
(např. pro medián je halda přibližně poloviční
– což ale znamená, že má jen o 1 hladinu méně)

4. QuickSelect (modifikace QuickSortu)

Základní idea:

- po rozdělení seznamu podle pivota pracujeme dál jen s tou částí, ve které je hledaný prvek (poznáme ji podle délky)

```
def quickselect(s, k):
    """výběr k-tého nejmenšího prvku ze seznamu s"""
    if len(s) <= 1: return s[0]
    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 ]
    a = len(vlevo)
    b = len(vlevo) + len(stred)

    if k < a:
        return quickselect(vlevo, k)
    elif k < b:
        return x
    else:
        return quickselect(vpravo, k-b)
```

Jiná implementace – nevytváří se nové seznamy:

- místo nových seznamů pracujeme s úseky původního seznamu
- po přerovnání prvků podle pivota pracujeme dál jen s tou částí, ve které je hledaný prvek, druhou část už není třeba dotřídit
- volba té správné části: ta, v níž leží *K*-tý nejmenší prvek, tzn. kde je index *K*-1
- výsledek výpočtu: prvek S[*K*-1]
- při realizaci není třeba rekurze ani zásobník, jediné rekurzivní volání se snadno nahradí jednoduchým cyklem

```
def quickselect(s, k):
    """výběr k-tého nejmenšího prvku ze seznamu s"""
    zac, kon = 0, len(s)-1
    while zac < kon:
        x = s[k-1]      #možná volba pivota, lze i jinak
        i, j = zac, kon
        while i <= j:
            while s[i] < x: i += 1
            while s[j] > x: j -= 1
            if i < j:
                s[i], s[j] = s[j], s[i]
                i += 1; j -= 1
            elif i == j:
                i += 1; j -= 1
        # úsek <zac,kon> rozdělen na <zac,j> a <i,kon>
        if k-1 < i: kon = j
        if k-1 > j: zac = i
    return s[k-1]
```

Časová složitost

- *v nejhorším případě:*
provede se plný QuickSort (jeho nejhorší případ), složitost **O(*N*^{2})**

- *v nejlepším případě (půlení):*
procházejí se po řadě úseky délky *N*, *N*/2, *N*/4, …, 1,
celkem vykonaná práce = součet jejich délek se blíží k *2N,*
tedy časová složitost algoritmu **O(*N*)**

- *v průměrném případě* rovněž **O(*N*)**

5. Lineární algoritmus

- obdobný postup jako v případě QuickSelectu,
jenom pivota, podle něhož rozdělujeme prvky v poli, nevolíme
náhodně, ale nějak chytře – aby byl natolik blízký mediánu,
že nám zaručí lineární časovou složitost **O(*N*)** i v nejhorším případě
(ale nemusí to být pokaždé přesně medián)

- jedna možná taková volba pivota *X*: medián z mediánů pětic
(zajišťuje i v nejhorším případě zahodit alespoň 3/10 prvků
z aktuálního úseku a to stačí)

Algoritmus:

- čísla rozdělit na pětice
- v každé pětici nalézt medián (triviální, konstantní složitost)
- mediány všech pětic $\rightarrow$ množina M
- určit medián množiny M $\rightarrow$ číslo *X*
  (rekurzivním voláním téhož algoritmu na menší množinu)
- toto *X* použít jako pivot v algoritmu QuickSelect

tedy:

- rozdělit čísla do množin $M_1=\{a_i;\ a_i<X\}$, $M_2=\{a_i;\ a_i=X\}$, $M_3=\{a_i;\ a_i>X\}$
- pokud $K \leq |M_1| \rightarrow$ nalézt *K*-té nejmenší číslo v množině M_1
  (rekurzivním voláním téhož algoritmu na menší množinu)
- pokud $K > |M_1|$ & $K \leq |M_1 \cup M_2| \rightarrow$ výsledkem je *X*
- pokud $K > |M_1 \cup M_2| \rightarrow$ nalézt $(K - |M_1 \cup M_2|)$ -té nejmenší číslo v množině M_3
  (rekurzivním voláním téhož algoritmu na menší množinu)

Časová složitost

označme
S_1 = {tři nejmenší prvky z pětic, jejichž medián je menší než *X*}
S_3 = {tři největší prvky z pětic, jejichž medián je větší než *X*}

z tranzitivity nerovnosti plyne, že $S_1 \subseteq M_1$ a $S_3 \subseteq M_3$

$|S_1| \approx |S_3| \approx 3/5$ z $1/2$ z $N = 3/10\ N$

proto $|M_1| \geq 3/10\ N$ a $|M_3| \geq 3/10\ N$

a jelikož množiny M_1 a M_1 jsou disjunktní, platí také

$\mathbf{|M_1| \leq 7/10\ N}$ a $\mathbf{|M_3| \leq 7/10\ N}$

t(N) – časová složitost výpočtu pro data velikosti *N*

$$t(N) \leq c.N + t(1/5\ N) + t(7/10\ N)$$

$c.N$	$t(1/5\ N)$	$t(7/10\ N)$
určení mediánů pětic	nalezení *X* (rekurze)	rekurzivní volání na M_1 nebo M_3

Indukcí dokážeme existenci konstanty *d* takové, $t(N) \leq d.N$
(neboli funkce *t* je v nejhorším případě lineární):

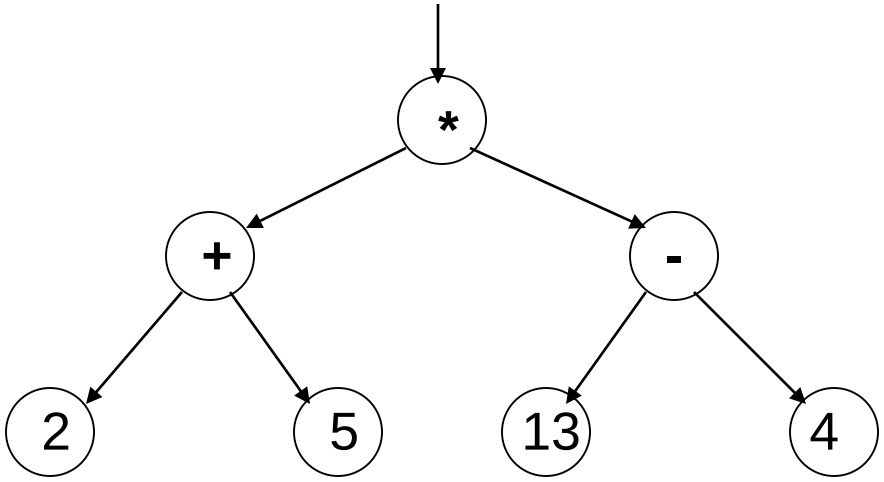
1. pro $N=1$ … triviální
2. dokážeme platnost pro *N*, když platí pro menší

$t(N) \leq c.N + d.1/5.N + d.7/10.N$ (podle indukčního předpokladu)
$t(N) \leq N.(c + 9/10.d)$
stačí zvolit takové *d*, aby $c + 9/10.d \leq d$, tedy $d \geq 10c$,
pro něj skutečně $\boldsymbol{t(N) \leq d.N}$

Reprezentace aritmetického výrazu

- binární strom reprezentující aritmetický výraz

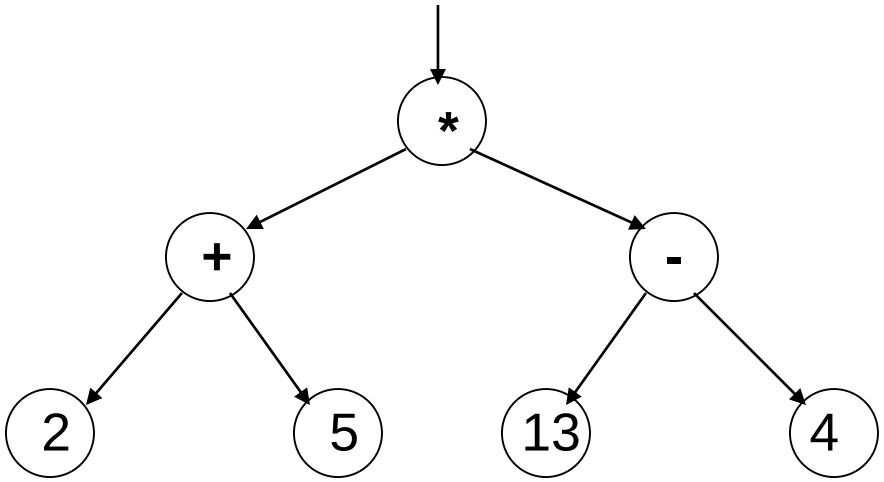
(2 + 5) * (13 - 4)



- listy stromu obsahují operandy (čísla)
- vnitřní uzly obsahují operátory (znaménka)
- závorky ve stromě nejsou,
  pořadí vyhodnocení je určeno strukturou stromu

```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x               # uložená hodnota
        self.levy = None            # levý syn
        self.pravy = None           # pravý syn
```




```
v = Vrchol('*')
v.levy = Vrchol('+')
v.levy.levy = Vrchol(2)
v.levy.pravy = Vrchol(5)
v.pravy = Vrchol('-')
v.pravy.levy = Vrchol(13)
v.pravy.pravy = Vrchol(4)

print(v.vyraz())
```

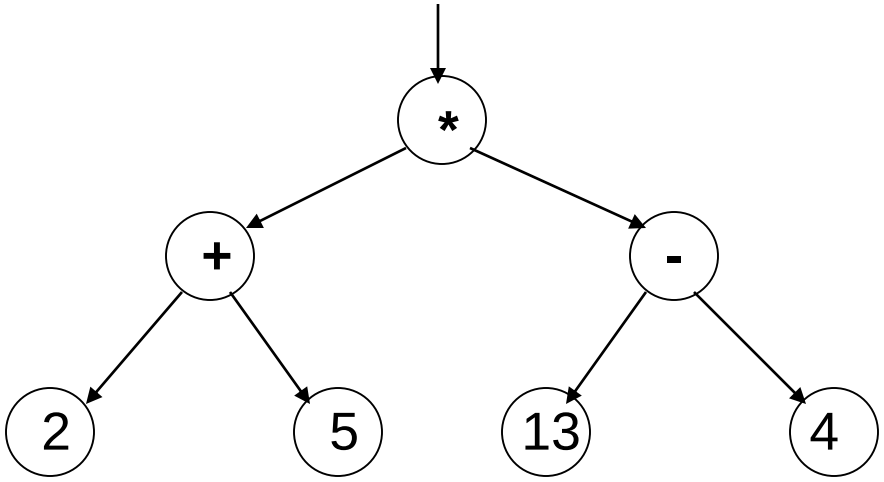
Vyhodnocení aritmetického výrazu reprezentovaného binárním stromem – rekurzívně (metoda Rozděl a panuj):

```
def vyraz(self):
    """vyhodnocení aritmetického výrazu reprezentovaného
       stromem s kořenem v tomto vrcholu
    """
    if self.levy == None:      # list
        return self.info
    elif self.info == '+':
        return self.levy.vyraz() + self.pravy.vyraz()
    elif self.info == '-':
        return self.levy.vyraz() - self.pravy.vyraz()
    elif self.info == '*':
        return self.levy.vyraz() * self.pravy.vyraz()
    elif self.info == '/':
        return self.levy.vyraz() / self.pravy.vyraz()
```

Notace aritmetického výrazu

- průchod binárním stromem reprezentujícím aritmetický výraz
- v navštívených uzlech vypisujeme uloženou hodnotu

(2 + 5) * (13 - 4)



průchod preorder → PREFIX	* + 2 5 - 13 4
průchod inorder → INFIX (bez závorek!)	2 + 5 * 13 - 4
průchod postorder → POSTFIX	2 5 + 13 4 - *

```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x              # uložená hodnota
        self.levy = None           # levý syn
        self.pravy = None          # pravý syn

    def preorder(self):
        """průchod stromem s kořenem v tomto vrcholu
           metodou preorder, vypisuje hodnoty všech
           vrcholů
        """
        print(self.info)
        if self.levy != None:
            self.levy.preorder()
        if self.pravy != None:
            self.pravy.preorder()
```

```
def inorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou inorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.inorder()
    print(self.info)
    if self.pravy != None:
        self.pravy.inorder()

def postorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou postorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.postorder()
    if self.pravy != None:
        self.pravy.postorder()
    print(self.info)
```

průchod preorder → PREFIX	* + 2 5 - 13 4
průchod inorder → INFIX (bez závorek!)	2 + 5 * 13 - 4
průchod postorder → POSTFIX	2 5 + 13 4 - *

- vždy stejné pořadí operandů – listy stromu procházíme ve všech případech zleva doprava (2 5 13 4)
- v prefixovém zápisu operátor bezprostředně předchází své dva argumenty (tzn. čísla nebo podvýrazy), v postfixovém je následuje
- v prefixovém a postfixovém zápisu výrazu nejsou závorky, pořadí vyhodnocování je plně určenou strukturou výrazu
- inorder průchod stromem vytvořil chybný infixový zápis bez závorek, z něhož není zřejmé správné pořadí vyhodnocování výrazu

Terminologická poznámka:
prefix = polská notace (Polish notation) – Łukasiewicz
postfix = reverzní polská notace (reverse Polish notataion, RPN)

Získání správného infixového zápisu výrazu:

```
    def infix(self):
        """průchod stromem s kořenem v tomto vrcholu
           metodou inorder, vypisuje hodnoty všech
           vrcholů
        """
        if self.levy == None:      # je to list
            print(self.info, end='')
        else:                      # není to list
            print('(', end='')
            self.levy.infix()
            print(self.info, end='')
            self.pravy.infix()
            print(')', end='')
```

Vyhodnocení výrazu v postfixové notaci

- snadné, využití např. dříve u kalkulaček, v překladačích
- jeden průchod zápisem výrazu zleva doprava
- používá zásobník na ukládání číselných hodnot

Postup zpracování postfixového zápisu:
číslo $\rightarrow$ vložit do zásobníku
znaménko $\rightarrow$ vyzvednout ze zásobníku horní dvě čísla
provést s nimi operaci určenou znaménkem
výsledek operace vložit do zásobníku
konec $\rightarrow$ na zásobníku je jediné číslo = hodnota výrazu

- pozor na pořadí operandů u nekomutativních operátorů
  (na vrcholu zásobníku je pravý operand, pod ním levý)
- časová složitost O(*N*), kde *N* je délka výrazu

```
class Stack:
    def __init__(self):
        ...
    def push(self, value):
        ...
    def pop(self):
        ...
    def count(self):
        ...

OPERATORS = {
    "+": (lambda a, b: a + b),
    "-": (lambda a, b: a - b),
    "*": (lambda a, b: a * b),
    "/": (lambda a, b: a // b)
}
```

```
def evaluate_postfix(expression):
    """
    vyhodnocení aritemetického výrazu v postfixu
    ve výrazu vše odděleno mezerami
    """
    parts = expression.split()
    stack = Stack()
    for part in parts:
        if part in OPERATORS.keys():
            arg1 = stack.pop()
            arg2 = stack.pop()
            result = OPERATORS[part](arg2, arg1)
            stack.push(result)
        else:
            stack.push(int(part))
    result = stack.pop()
    assert stack.count() == 0
    return result
```

Vyhodnocení výrazu v prefixové notaci

1. možnost:

- průchod výrazem **odzadu**, postup jako u postfixu
- pouze se změní pořadí operandů při zpracování znaménka:
  při vyzvednutí ze zásobníku je na vrcholu zásobníku levý operand, pod ním je pravý
- časová složitost O(*N*), kde *N* je délka výrazu

2. možnost:
- jeden průchod zápisem výrazu zleva doprava
- **zásobník** na ukládání znamének a číselných hodnot

Postup zpracování prefixového zápisu odpředu:
- znaménko nebo číslo → vložit do zásobníku
- když se tím na vrcholu zásobníku sejdou dvě čísla → vyzvednout je ze zásobníku, dále vyzvednout znaménko uložené pod nimi, provést s čísly operaci určenou znaménkem a výsledek operace vložit do zásobníku (což může opětovně vyvolat tentýž proces vyhodnocení)
- konec → na zásobníku je jediné číslo = hodnota výrazu

- pozor na pořadí operandů u nekomutativních operátorů (na vrcholu zásobníku je pravý operand, pod ním levý)
- časová složitost O(*N*), kde *N* je délka výrazu

3. možnost: **rekurze**

- rekurzivní funkce na vyčíslení prefixového zápisu od zadaného indexu
- globálně udržujeme pozici indexu

- když je prvním znakem výrazu číslice, výrazem je jen jedno číslo
  $\rightarrow$ funkce vrátí jeho hodnotu (a posune index za něj)
- když je prvním znakem znaménko Z, funkce posune index za něj, potom provede dvě rekurzivní volání sebe sama a s výsledky těchto volání vykoná operaci určenou znaménkem Z

- celkem se provede jeden průchod zápisem výrazu zleva doprava
- časová složitost O(*N*), kde *N* je délka výrazu

Převod infix → postfix

máme zadán aritmetický výraz v běžné infixové notaci,
chceme ho převést do postfixové notace

- provede se jeden průchod zápisem výrazu zleva doprava,
  tedy časová složitost O(*N*)
- používá zásobník na ukládání znamének
- v postfixovém zápisu jsou čísla ve stejném pořadí jako v infixovém,
  znaménka je proto třeba pozdržet na zásobníku, aby se dostala
  na správné místo až za svoje argumenty

Postup zpracování infixového zápisu:

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

spojení dvou předchozích algoritmů:

- převod výrazu z infixu do postfixu v čase O(*N*)
- vyhodnocení postfixové notace v čase O(*N*)

→ celková časová i paměťová složitost **O(*N*)**

obě fáze výpočtu se mohu provádět

- buď postupně (s uložením vytvořené postfixové notace výrazu)
- nebo souběžně (tzn. vznikající postfixová notace se neukládá, ale rovnou se průběžně vyhodnocuje)
  → algoritmus používá dva zásobníky – jeden na znaménka a druhý na čísla

Postavení aritmetického binárního stromu ze zápisu výrazu

postfixová nebo prefixová notace
- algoritmus podobný jako při vyhodnocování výrazu,
do zásobníku se vždy ukládá odkaz na nově vytvořený uzel,
místo provádění operací se uzly s operandy zapojují pod uzel
s operátorem jako jeho synové

infixová notace
- nejprve výraz převedeme do postfixové notace,
z té pak postavíme aritmetický strom