Konstrukce haldy v lineárním čase

- výchozí rozložení dat představuje úplný binární strom hloubky *d* (bez uspořádání hodnot do haldy)
- nejprve postavíme „haldy“ z podstromů, jejichž kořeny mají hloubku *d*-1, potom pro *d*-2, … atd., až do kořene celé haldy
- stavění hald se provádí záměnami hodnot od kořene k listům (výměna s menším z obou synů)

Časová složitost

hloubka	počet uzlů	max. počet výměn pro každý z nich
0	2^0	d
1	2^1	d-1
…	…	…
j	2^{j}	d-j
…	…	…
d-1	2^{d-1}	1

Pozorování: Při tomto postupu konstrukce haldy má hodně uzlů malý maximální počet výměn a jen málo z nich může absolvovat výměn hodně → celková časová složitost **O(*N*)**.

Celkem se provede výměn (viz tabulka):

$$\sum_{j=0}^{d-1} 2^j (d-j) = d\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j.2^j =$$

$$= d.(2^d - 1) - ((d-2).2^d + 2) = O(2^d) = O(N)$$

… viz dva důkazy matematickou indukcí dále

Důkaz matematickou indukcí č.1: $\sum_{j=0}^{d-1} 2^j = 2^d - 1$

1. pro $d = 1$ zjevně platí

2. nechť platí pro všechna $d < D$, dokazujeme platnost pro $D > 1$:

$$\sum_{j=0}^{D-1} 2^j = \sum_{j=0}^{D-2} 2^j + 2^{D-1} = \underset{\substack{\uparrow \\ \text{podle indukčního předpokladu}}}{\left(2^{D-1} - 1\right)} + 2^{D-1} = 2.2^{D-1} - 1 = 2^D - 1$$

qed

Důkaz matematickou indukcí č.2: $$\sum_{j=0}^{d-1} j\,.2^{j} = (d-2).2^{d} + 2$$

1. pro d=1 zjevně platí

2. nechť platí pro všechna $d < D$, dokazujeme platnost pro $D > 1$:

$$\sum_{j=0}^{D-1} j\,.2^{j} = \sum_{j=0}^{D-2} j\,.2^{j} + (D-1).2^{D-1} = \underbrace{\left((D-3).2^{D-1} + 2\right)}_{\uparrow} + (D-1)\,.2^{D-1} =$$

podle indukčního předpokladu

$$= D\,.2^{D-1} - 3.2^{D-1} + D\,.2^{D-1} - 2^{D-1} + 2 = 2.D\,.2^{D-1} - 4.2^{D-1} + 2 =$$

$$= D\,.2^{D} - 2.2^{D} + 2 = (D-2).2^{D} + 2$$ qed

Třídění haldou (haldové třídění, HeapSort)

- z prvků postavit haldu (*N* x přidání prvku do haldy)
  $\rightarrow$ časová složitost O(*N*.log *N*)
  nebo zdola v lineárním čase O(*N*)
- haldu postupně rozebrat (*N* x odebrat minimum z haldy)
  $\rightarrow$ časová složitost O(*N*.log *N*)

$\rightarrow$ tedy celková časová složitost **O(*N*.log *N*)** i v nejhorším případě

- třídí „na místě“ (tzn. nepotřebuje další datovou strukturu velikosti *N*):
prvky uložené v poli postupně řadí do haldy, přičemž haldu staví
v levé části téhož pole
pak rozebírá postupně haldu a vyřazované prvky ukládá postupně do
pravé části téhož pole, kde se uvolňuje místo po zkracující se haldě

Prioritní fronta

- podobné jako fronta, prvky se „předbíhají“ podle svých priorit
- zachování vzájemného pořadí mezi prvky téže priority požadovat můžeme, ale nemusíme (záleží na konkrétní aplikaci)

Možnosti implementace:

- seznam (pole, LSS), do něhož zařazujeme podle priority
- seznam (pole, LSS), z něhož vybíráme podle priority
- samostatné seznamy pro každou hodnotu priority (pokud tyto hodnoty známe a není jich mnoho)
- halda řazená podle priorit – pokud nepožadujeme zachovat pořadí
- halda řazená podle dvojic (priorita, čas příchodu) – pokud požadujeme zachovat vzájemné pořadí mezi prvky téže priority

Slovník (dictionary)

- uchovává dvojice *klíč – hodnota* (klíč je jednoznačný)
- „asociativní pole“
- uložené údaje se vyhledávají podle klíče (indexuje se klíčem)
- klíč může mít jakoukoliv neměnitelnou hodnotu, dokonce třeba každý záznam má klíč jiného typu
- některé programovací jazyky přímo podporují
- efektivní implementace je složitější (hešování, případně vyhledávací stromy)

Rekurze

- objekt je definován pomocí sebe sama

V programování ve dvou hladinách:

- **rekurzivní algoritmus** – řešení úlohy je definováno pomocí řešení menších instancí téhož problému (tzn. podúloh stejného charakteru)

- **rekurzivní volání funkce** – funkce volá sama sebe (přímo nebo případně nepřímo prostřednictvím jiných funkcí)

Většinou se rekurzivní algoritmy realizují pomocí rekurzivních volání, ale není to nezbytné:

- rekurzivní algoritmus lze realizovat bez rekurzivních volání (pomocí vlastního zásobníku na uložení rozpracovaných nedokončených podúloh)
$\rightarrow$ více práce pro programátora, program obvykle delší a méně přehledný, výpočet ale může být o něco efektivnější

- rekurzivní volání lze teoreticky použít i při realizaci nerekurzivních iteračních algoritmů (dokonce každý cyklus lze nahradit rekurzivní procedurou)
$\rightarrow$ většinou nevhodné, nečitelný a méně efektivní program

Ukončení rekurze

- rekurzivní volání funkce musí být vázáno na nějakou podmínku, která časem jistě přestane platit → jinak „zacyklení výpočtu“

- na rozdíl od nekonečného while-cyklu dojde v Pythonu k zastavení výpočtu po dosažení maximální povolené hloubky rekurze (cca 1000)

```
def rekurze(p):
    print(p)
    rekurze(p+1)

rekurze(1)
```

Poznámka: programovací jazyky bez limitu na hloubku rekurze
→ běhová chyba přetečení zásobníku (stack overflow)

Průběh výpočtu při rekurzivním volání funkce

- najednou je rozpočítáno více exemplářů téže funkce

- všechny počítají podle téhož kódu

- každý exemplář má na volacím zásobníku svůj vlastní aktivační záznam s lokálními proměnnými, parametry a technickými údaji (kde je rozpočítán, návratová adresa)

- funkce nemá přístup k lokálním proměnným jiného rekurzivního exempláře

Příklad 3: výpis znaků ze vstupu pozpátku

```
def otoc():
    u = input("Znak: ")
    if u != " ":
        otoc()
    print(u)
```

Vstup:	A	*Výstup:*	<mezera>
	B		C
	C		B
	<mezera>		A

Příklady jednoduchých rekurzivních funkcí

Palindrom – řetězec se čte stejně zleva i zprava (je symetrický)

```
def palindrom1(s):
    n = len(s)
    for i in range(n//2):
        if s[i] != s[n-i-1]:
            return False
    return True

def palindrom2(s):
    if len(s) <=1 :
        return True
    else:
        return s[0] == s[-1] and palindrom2(s[1:-1])
```

Eukleidův algoritmus (odčítací) realizovaný cyklem *(bylo dříve):*

```
def nsd(x, y):
    while x != y:
        if x > y:
            x -= y
        else:
            y -= x
    return x
```

Eukleidův algoritmus realizovaný rekurzivní funkcí
- přesně kopíruje rekurzivní vztah pro NSD,
na němž je Eukleidův algoritmus založen:

když X > Y   NSD(X ,Y) = NSD(X-Y, Y)
když X < Y   NSD(X ,Y) = NSD(X, Y-X)
když X = Y   NSD(X ,Y) = X

```
def nsd(x, y):
    if x > y:
        return nsd(x-y, y)
    elif x < y:
        return nsd(x, y-x)
    else:
        return x
```

Faktoriál *n!* (součin čísel od 1 do *n*, pro *n* >= 0)

rekurzivní definice: $n! = 1$ pro $n = 0$
$n! = n \,.\, (n-1)!$ pro $n > 0$

```
def faktorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

def faktorial(n):
    if n == 0:
        return 1
    else:
        return n * faktorial(n-1)
```

v obou případech časová složitost O(*n*)

Fibonacciho čísla $F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$ pro $n > 1$

rekurzivní definice posloupnosti čísel
→ realizace rekurzivní funkcí přesně podle definice:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

funkce teoreticky správná, ale časová složitost exponenciální
→ pro n > cca 40 prakticky nepoužitelná
důvod: mnohokrát se opakovaně počítají stejné věci

Možnosti řešení:

1. rekurzivní algoritmus + pomocné pole velikosti *n* pro uložení již spočítaných funkčních hodnot

   $\rightarrow$ každé F_i se počítá jen jednou $\rightarrow$ časová složitost O(n)

   „chytrá rekurze"
   kešování hodnot (cache = mezipaměť)
   memoizace (memory = paměť)
   dynamické programování

2. počítat hodnoty iteračně odspodu – v pořadí F_1, F_2, … F_n
$\rightarrow$ časová složitost $O(n)$, navíc stačí konstantní paměť

dynamické programování

```
def fib(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        a, b = b, a+b
        n -= 1
    return b
```

3. z rekurzivní definice odvodit explicitní vzorec a počítat podle něj

$$F_n = \frac{\sqrt{5}}{5}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right)$$

4. využití rychlého umocňování matice

Platí rovnost $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$

Tedy $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} F0 \\ F1 \end{pmatrix} = \begin{pmatrix} F1 \\ F2 \end{pmatrix}$

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} F1 \\ F2 \end{pmatrix} = \begin{pmatrix} F2 \\ F3 \end{pmatrix}$$

…

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n . \begin{pmatrix} F0 \\ F1 \end{pmatrix} = \begin{pmatrix} Fn \\ Fn+1 \end{pmatrix}$$

Matici $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$ spočítáme rychlým umocňováním

$\rightarrow$ časová složitost O(log *N*)