Algoritmus řazení slučováním (funkc. verze)

```
mergeSort      ::  [Int] → [Int]
mergeSort []    =  []
mergeSort [x]   =  [x]
mergeSort s     =  merge (mergeSort u) (mergeSort v)
                   where (u,v) = splitAt (n `div` 2) s
                         n     = length s

merge s []          =  s
merge [] t          =  t
merge (x:u) (y:v)   =  if x ≤ y  then  x : merge    u   (y:v)
                                 else  y : merge (x:u)    v
```

Příklad:

ms [4,3,1,6,7,2,8,5]

mr (*ms* [4,3,1,6]) (*ms* [7,2,8,5])

mr (*mr* (*ms* [4,3]) (*ms* [1,6])) (*mr* (*ms* [7,2]) (*ms* [8,5]))

mr (*mr* (*mr* (*ms* [4])(*ms* [3])) (*mr* (*ms* [1])(*ms* [6]))) (*mr* (*mr* (*ms* [7])(*ms* [2])) (*mr* (*ms* [8])(*ms* [5])))

mr (*mr* (*mr* [4] [3]) (*mr* [1] [6])) (*mr* (*mr* [7] [2]) (*mr* [8] [5]))

mr (*mr* [3,4] [1,6]) (*mr* [2,7] [5,8])

mr [1,3,4,6] [2,5,7,8]

[1,2,3,4,5,6,7,8]

Věta: Nechť *In* $=$ *Out* je množina všech seznamů (posloupností) celých čísel, $\varphi(s) \equiv s$ je konečný, $\psi(s,t) \equiv t$ je permutací seznamu s a t je neklesající. Pak `mergeSort` je totálně korektní vzhledem k podmínkám φ, ψ.

Lemma: Časová složitost pomocné funkce `merge` je $T_{\texttt{merge}} \in \Theta(n)$.

Věta: Časová složitost funkce `mergeSort` je $T_{\texttt{mergeSort}} \in \Theta(n \cdot \log n)$.

Důkaz totální korektnosti: Konvergence i parciální korektnost se dokáže indukcí podle délky seznamu s, parciální korektnost pomocné funkce `merge` se dokáže indukcí podle součtu délek obou slučovaných seznamů.

Lineární složitost funkce `merge` je zřejmá: konstantní délka porovnání a připojení menšího prvku na začátek posloupnosti se dohromady provede tolikrát, kolik je součet délek obou vstupních posloupností (argumentů funkce `merge`).

Odvození složitosti funkce `mergeSort` je na následujících stranách.

Algoritmus řazení slučováním (imp. verze)

```
type Elem = Integer;  Posl = array [1..MAX] of Elem;

procedure mergeSort (n:Integer; var A:Posl);

  var B : Posl; { pomocné pole }

  procedure msort (p,r:Integer); { seřadí pole od p do r }
     var q : Integer;
     begin if p < r then begin
                            q := ⌊(p+r)/2⌋ ;
                            msort (p,q) ;
                            msort (q+1,r) ;
                            merge (p,q,r)
                         end
     end
```

```
procedure merge (p,q,r:Integer);
   { sloučí úsek A[p],...,A[q] s úsekem A[q+1],...,A[r] }
   var i,j : Integer;
   begin
     i := p ;  j := q + 1 ;
     for k := p to r do
       if        (i≤q) && (j≤r) && (A[i]≤A[j]) || (j>r)
         then     begin B[k] := A[i] ; i := i+1 end
         else if (i≤q) && (j≤r) && (A[i]>A[j]) || (i>q)
         then     begin B[k] := A[j] ; j := j+1 end ;
     for k := p to r do  A[k] := B[k]
   end


begin { mergeSort }
   msort (1,n)
end   { mergeSort }
```

Poznámka: Podmíněný příkaz v těle cyklu for procedury `merge` lze ekvivalentně zapsat kratčeji takto (dokažte)

```
if ( i ≤ q ) && ( j > r || A[i] ≤ A[j] )
   then  begin B[k] := A[i] ; i := i+1 end
   else  begin B[k] := A[j] ; j := j+1 end ;
```

Časová složitost algoritmu řazení slučováním

Analýza nejhoršího případu

Pro danou velikost n je délka výpočtu vždy stejná.

Časová složitost

$$T(1) = a$$

$$T(n) = b + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + D(n), \quad D \in \Theta(n)$$

$$T'(1) = a$$

$$T'(n) = b + 2\,T'(n/2) + n \cdot c, \quad T' \in \Theta(T).$$

Analýza nejhoršího případu je zde triviální: pro každou vstupní posloupnost délky n algoritmus dělá stejnou práci. Pro složitost $T : \mathbb{N} \to \mathbb{N}$ tedy platí, že $T(n)$ je rovno délce výpočtu na *libovolné* vstupní posloupnosti délky n.

Je-li $n \leq 1$, pak $T(n) = a$, kde a je konstanta.

Je-li $n > 1$, pak $T(n) = b + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + D(n)$, kde D je složitost zbytku těla procedury `msort` (bez rekursivního volání), tedy zejména pomocné procedury `merge`.

Procedura `merge` je tvořena cyklem s pevným počtem opakování (n), takže $D(n) = c \cdot n$, kde c je konstanta. Tedy $D \in \Theta(n)$.

Protože $\lfloor n/2 \rfloor$ a $\lceil n/2 \rceil$ se liší nejvýše o 1 (tj. o konstantu), můžeme tento rozdíl zanedbat. Takto upravená funkce T' roste stejně rychle jako T:
$T'(1) = a$
$T'(n) = b + 2\,T'(\lceil n/2 \rceil) + n \cdot c, \quad T' \in \Theta(T).$

Pro několik hodnot n dostáváme

$$\begin{array}{lcll}
T'(1) & = & a & \\
T'(2) & = & b + 2 \cdot T'(1) + 2c & = \quad 2a + b + 2c \\
 & \vdots & & \\
T'(4) & = & b + 2 \cdot T'(2) + 4c & = \quad 4a + 3b + 8c \\
 & \vdots & & \\
T'(8) & = & b + 2 \cdot T'(4) + 8c & = \quad 8a + 7b + 24c \\
 & \vdots & & \\
T'(16) & = & b + 2 \cdot T'(8) + 16c & = \quad 16a + 15b + 64c \\
 & \vdots & & \\
T'(32) & = & b + 2 \cdot T'(16) + 32c & = \quad 32a + 31b + 160c \\
 & \vdots & &
\end{array}$$

Všimneme-li si hodnot T' na mocninách dvou, můžeme vyslovit tvrzení

Věta: Pro každé $k \in \mathbb{N}$ je

$$T'(2^k) = 2^k a + (2^k - 1)b + 2^k kc$$

Vyjádřeno pomocí n:

$$T'(n) = a \cdot n + b \cdot (n - 1) + c \cdot n \log_2 n$$

Protože lineární funkce $a \cdot n$, $b \cdot n$ rostou nejvýše tak rychle jako $n \cdot \log n$, tak $T' \in \Theta(n \log n)$.

T roste stejně rychle, takže i $T \in \Theta(n \log n)$.

Rovnosti $T(1) = c$
$T(n) = 2\,T(\lceil n/2 \rceil) + d \cdot n$
jsou speciálním případem situace, v níž je
$T(1) = c$
$T(n) = a\,T(\lceil n/b \rceil) + f(n)$, kde $a \geq 1, b > 1, c > 0$ a f je kladná funkce. Například pro algoritmus řazení slučováním je $a = 2, b = 2, f \in \Theta(n)$, tedy případ „2“ následující věty.

Věta: Nechť $a \geq 1, b > 1,$ f je kladná funkce a
$T(1) = c$
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$
Potom:
1. Pokud $f \in O(n^{\log_b a - \varepsilon})$ pro nějaké $\varepsilon > 0$, pak $T \in \Theta(n^{\log_b a})$.
2. Pokud $f \in \Theta(n^{\log_b a})$, pak $T \in \Theta(n^{\log_b a} \log n)$.
3. Pokud $f \in \Omega(n^{\log_b a + \varepsilon})$ pro nějaké $\varepsilon > 0$ a pokud $a \cdot f(n/b) \leq d \cdot f(n)$ pro nějaké $d < 1$ a skoro všechna n, pak $T \in \Theta(f(n))$.

Dolní odhad složitosti řadicích algoritmů

Def: *Asociativní řadicí algoritmus* je algoritmus, jehož množinu přípustných vstupních dat tvoří všechny konečné posloupnosti celých čísel, a takový, že jeho výsledkem je neklesající permutace vstupní posloupnosti.

Poznámka: Asociativní řadicí algoritmus tedy nemůže předpokládat nic o velikosti řazených prvků a jedinou možností testování je srovnávat prvky mezi sebou.

Věta: Každý sekvenční asociativní řadicí algoritmus má složitost v $\Omega(n \log n)$.

Poznámka: Věta říká, že *dolní odhad* (časové) složitosti problému asociativního řazení je v množině $\Omega(n \log n)$. Protože však známe konkrétní řadicí algoritmy s touto složitostí, vidíme, že tento odhad je dokonce *těsný*: Časová složitost problému asociativního řazení je v množině $\Theta(n \log n)$.

Např. složitost řazení vkládáním, výběrem, bubláním či rozdělováním je $\Theta(n^2) \subseteq \Omega(n \log n)$, složitost řazení slučováním či haldou je $\Theta(n \log n) \subseteq \Omega(n \log n)$.

Pro určení složitosti uvažujeme zápis algoritmu v jednoduchém funkcionálním nebo imperativním jazyce, tj. ekvivalentní *sekvenční* implementaci RAMem.

Důkaz Věty o dolním odhadu složitosti

Bez újmy na obecnosti můžeme předpokládat, že prvky řazené posloupnosti jsou navzájem různé: chceme-li nejhorší případ, musíme seřazovat co nejvíce prvků. Dokonce se můžeme omezit jen na permutace množiny $\{1, \ldots, n\}$: asociativní algoritmus si všímá jen srovnání dvojic prvků.

Poněvadž existuje celkem $n!$ možných permutací a pro každou takovou permutaci na vstupu musí podle téhož algoritmu proběhnout jiný výpočet, existuje aspoň $n!$ různých výpočtů. Ve všech těchto výpočtech je aspoň $n! - 1$ binárních testů, jejichž výsledky odliší jednotlivé výpočty. Ale to znamená, že v *nejdelším* výpočtu (jde nám o nejhorší případ) musí být aspoň $\lfloor \log_2 n! \rfloor$ testů. Pro složitost T *každého* algoritmu tedy musí platit $T(n) \geq \lfloor \log_2 n! \rfloor$. To znamená, že $T \in \Omega(\log n!) = \Omega(n \log n)$

Cvičení: V důkazu věty o dolním odhadu složitosti řadicích algoritmů se mlčky předpokládá, že složitost jednoho porovnání dvou čísel a_i a a_j je konstantní. Můžete uvést lepší (realističtější) odhad časové složitosti jednoho porovnání?

Cvičení: Ukažte, že $\log(n \cdot \log n) \in \Theta(\log n)$.

Cvičení: Uvážíme-li realistickou složitost jednoho porovnání, jaký vyjde dolní odhad složitosti řadicích algoritmů? Je v rozporu s dokázanou větou? Je jejím zesílením?

Algoritmus řazení rozdělováním

(funkcionální verze)

```
quickSort :: [Int] → [Int]
quickSort []     =  []
quickSort (p:t) =  quickSort lt ++ [p] ++ quickSort ge
                   where lt = [ x | x←t, x<p ]
                         ge = [ x | x←t, x≥p ]
```

Věta: Nechť *In* $=$ *Out* je množina všech seznamů (posloupností) celých čísel,

$\varphi(s) \equiv$ „s je konečný“,

$\psi(s,t) \equiv$ „t je permutací seznamu s a t je neklesající“.

Pak `quickSort` je totálně korektní vzhledem k podmínkám φ, ψ.

Věta: Časová složitost algoritmu `quickSort` je v $\Theta(n^2)$.

Důkaz korektnosti: Konvergence i parciální korektnost se ukáže indukcí vzhledem k délce seznamu s.

Řazení rozdělováním (imperativní verze)

```
type  Element = Integer;
      Posl = array [1..MAX] of Element;

procedure QuickSort (n:Integer, var A:Posl);

  procedure Q (l,r:Integer);
    var p: Integer;
    begin if l < r then begin p := Part (l,r);
                              Q (l,p-1);  Q (p+1,r)
                        end
    end;

  procedure Part (l,r:Integer);
    var i,j: Integer; x,y: Element;
    begin x := A[r];  i := l - 1;
          for j:=l to r-1 do
              if A[j] ≤ x then begin i := i+1;
                                     y := A[i];  A[i] := A[j];  A[j] := y;
                               end;
          y := A[i+1];  A[i+1] := A[r];  A[r] := y
          Part := i + 1
    end;

begin  Q (1,n)  end
```

Řazení rozdělováním (modif. imp. verze)

```
type  Element = Integer;
      Posl = array [1..MAX] of Element;

procedure QuickSort (n:Integer, var A:Posl);

  procedure Q (l,r:Integer);
    var p: Integer;
    begin if l<r then begin p := Part (l,r);
                            if p-l<r-p then begin Q (l,p-1);  Q (p+1,r) end
                                        else begin Q (p+1,r);  Q (l,p-1) end
                      end
    end;

  procedure Part (l,r:Integer);
    var i,j: Integer; x,y: Element;
    begin x := A[r];  i := l - 1;
          for j:=l to r-1 do
              if A[j]≤x then begin i := i+1;
                                   y := A[i];  A[i] := A[j];  A[j] := y;
                             end;
          y := A[i+1];  A[i+1] := A[r];  A[r] := y
          Part := i + 1
    end;

begin  Q (1,n)  end
```

Věta: Algoritmus řazení rozdělováním má složitost v množině $\Theta(n^2)$.

Poznámka: *Průměrná* délka výpočtu podle algoritmu řazení rozdělováním (tzv. *průměrná* nebo *očekávaná časová složitost*) je v množině $\Theta(n \log n)$.

Důkaz věty o kvadratické složitosti: Není těžké zjistit, že délka výpočtu cyklu (repeat) je $c \cdot n$ (celé pole se projde jedním průchodem a výměna prvků trvá konstantní dobu).

Označíme-li $T(n)$ složitost seřazení pole délky n, dostaneme

$$T(n) = c \cdot n + \text{max}\{T(q) + T(n - q - 1) \mid 0 \leq q < n\}$$

Při nevyváženém dělení posloupnosti na podposloupnosti délek $n - 1, 0$ je délka výpočtu kvadratická. Tedy $T \in \Omega(n^2)$. Ale výraz $T(q) + T(n - q - 1)$ nabude maxima pro $q = 0$ nebo $q = n - 1$ (o tomto případu víme, že je kvadratický) – druhá derivace $T(n)$ vzhledem ke q pro kvadratický odhad $T(n) \leq cn^2$ je kladná.
Dohromady, $T \in \Omega(n^2)$, $T \in O(n^2)$, tj. $T \in \Theta(n^2)$.

Neformální zdůvodnění poznámky o průměrné složitosti:
Při každém vyvolání procedury Q se posloupnost délky n rozdělí na dva úseky délek q a $n-q-1$. Předpokládáme-li rovnoměrné rozložení čísel (položek) v seřazované posloupnosti, pak pravděpodobnost, že $q < rn$, je rovna číslu r, $0 \leq r \leq 1$. Tedy pravděpodobnost, že se při výpočtu „nepříznivá“ hodnota hranice q vybere ve významném počtu případů, klesá s číslem r k nule. Proto stačí uvažovat jen případy, kdy se posloupnost dělí na úseky délek větších než n/r_0 pro nějaké pevné $r_0 > 0$ (ostatní případy mají nevýznamnou pravděpodobnost). Ale pak je největší hloubka zanoření rovna $\log_{\frac{1}{r_0}} n = -\log_{r_0} n$. Součet délek seřazovaných úseků v každé hloubce je nejvýše n, takže *průměrná* délka výpočtu je nejvýše $-c \cdot n \cdot \log_{r_0} n$, tedy v $O(n \log n)$. Dále se snadno ověří (například pro $r_0 = 1/2$), že $n \log n$ je i dolním odhadem, takže průměrná délka výpočtu je v $\Theta(n \log n)$.

Prostorová složitost řadicích algoritmů

Definujeme *prostorovou složitost* algoritmu analogicky jako časovou složitost, ale za míru složitosti místo počtu kroků výpočtu zvolíme maximální množství paměti, která bude během výpočtu obsazena.

Def: Prostorová složitost algoritmu A je funkce, která pro každou velikost vstupních dat je rovna velikosti obsazené paměti při výpočtu, jenž této paměti spotřebuje nejvíc.

Poznámka: Označíme-li $\rho(A, \sigma_x)$ množství paměti spotřebované výpočtem podle algoritmu A z počátečního stavu σ_x (odpovídajícího umístění dat x na vstupu), potom prostorová složitost algoritmu A je

$$S_A(n) = \max\{\rho(A, \sigma_x) \mid |x| = n\}$$

Prostorová složitost všech zmiňovaných řadicích algoritmů je lineární, tj. $S \in \Theta(n)$. Důvod je ten, že součástí dat, s nimiž algoritmus pracuje, je sama řazená posloupnost, a ta má délku n. Ostatní data (mimo tuto posloupnost, tj. „extrasekvenční“) mají v uváděných algoritmech velikost nejvýše $O(n)$.

Def: Zavedeme *extrasekvenční* prostorovou složitost řadicího algoritmu jako funkci zobrazující délky vstupní posloupnosti na velikost paměti obsazené při výpočtu v nejhorším případě, ale *nezapočítáváme paměť obsazenou seřazovanou posloupností*.

Řadicí algoritmy, které mají extrasekvenční prostorovou složitost konstantní, se nazývají *in situ*.

Řazení rozdělováním není *in situ*.

Věta: První varianta algoritmu `quickSort` má extrasekvenční prostorovou složitost $\Theta(n)$, druhá (modifikovaná) varianta tohoto algoritmu má extrasekvenční prostorovou složitost $\Theta(\log n)$.

Poznámka: Hovořit o řadicích algoritmech, zda jsou in situ, a zavádět jejich extrasekvenční prostorovou složitost má praktický význam jen tehdy, je-li posloupnost reprezentována polem a uložena v souvislém úseku paměti. Při jiných datových reprezentacích posloupnosti (např. seznamem) se extrasekvenční prostorová složitost nezavádí.

Řazení haldou

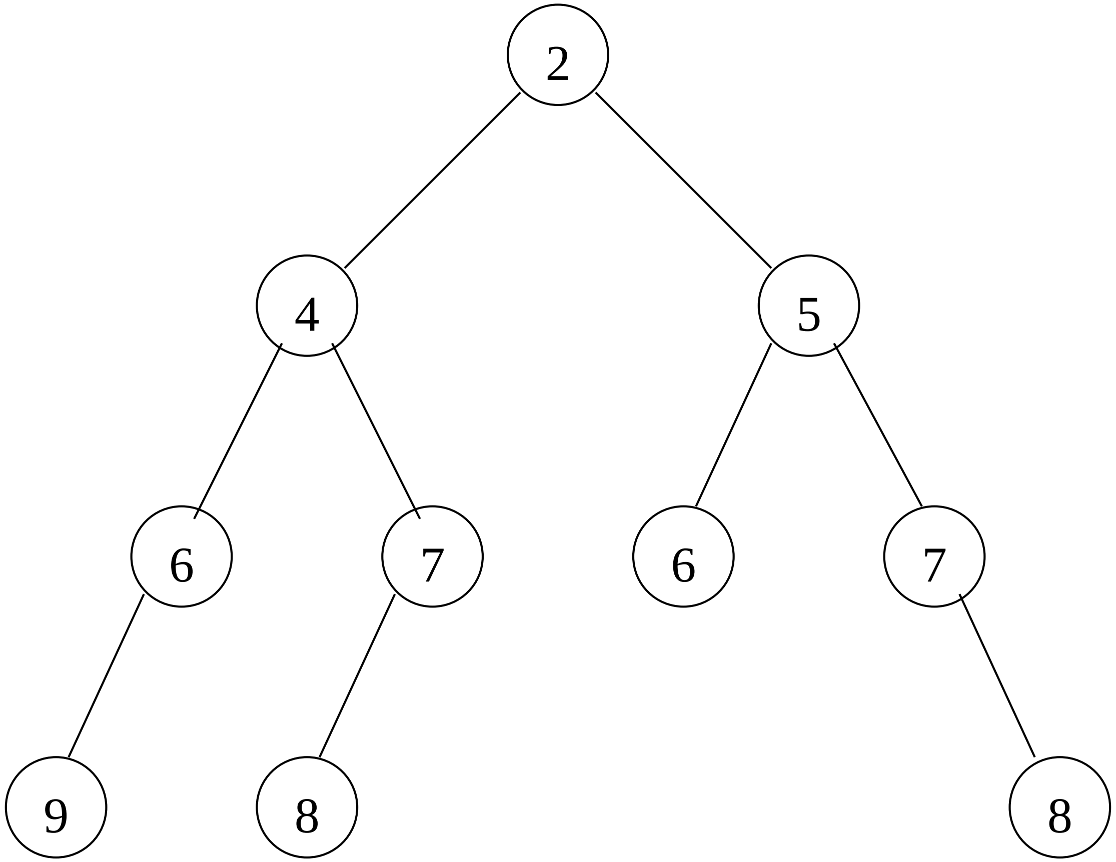
Def: Nechť K je úplně uspořádaná množina tzv. *klíčů* (typicky množina čísel s přirozeným uspořádáním $\leq$). *Binární halda* je binární strom, jehož uzly jsou ohodnoceny prvky množiny K, a který splňuje tyto vlastnosti:

1. Délky všech větví se liší nejvýše o 1: mají délku k, případně $k - 1$. Číslu k pak říkáme *hloubka* haldy.
2. Hodnoty uzlů na každé větvi jsou vzestupně (sestupně) uspořádány.

Jsou-li hodnoty uzlů na každé větvi seřazeny vzestupně (čím dále od kořene, tím větší hodnoty), je v kořenu haldy nejmenší prvek. Hovoříme pak o *minimové* haldě (*min-heap*).

Jsou-li hodnoty uzlů na každé větvi seřazeny sestupně (čím dále od kořene, tím menší hodnoty), je v kořenu haldy největší prvek. Hovoříme pak o *maximové* haldě (*max-heap*).

Ve funkcionální implementaci algoritmu řazení haldou, kde se pracuje s explicitní haldou (datovou strukturou binární strom), použijeme minimovou haldu. Naopak v imperativní implementaci, kde se pracuje s implicitní haldou (reprezentovanou polem), je výhodnější použít maximovou haldu.


2
4
5
6
7
6
7
9
8
8

Operace s binární haldou

```
data Heap = Empty | Node Elem Heap Heap
```

Základní operace – konstruktory a selektory – mají konstantní složitost.

Konstruktory

$\texttt{Empty} : \texttt{Heap}$

$\texttt{Node} : \texttt{Elem} \to \texttt{Heap} \to \texttt{Heap} \to \texttt{Heap}$

Selektory

$\texttt{rootVal} : \texttt{Heap} \dashrightarrow \texttt{Elem}$

$\texttt{leftHeap} : \texttt{Heap} \dashrightarrow \texttt{Heap}$

$\texttt{rightHeap} : \texttt{Heap} \dashrightarrow \texttt{Heap}$

$\texttt{isEmpty} : \texttt{Heap} \to \texttt{Bool}$

Další operace pro práci s binární haldou

Vyhledání minimálního prvku (v minimové haldě)

$\mathtt{minH} : \mathtt{Heap} \dashrightarrow \mathtt{Elem}$

$\mathtt{minH} = \mathtt{rootVal}$

(složitost $\Theta(1)$)

Odstranění minimálního prvku (z minimové haldy)

$\mathtt{extractMinH} : \mathtt{Heap} \dashrightarrow \mathtt{Heap}$

(složitost $\Theta(\log n)$)

Odstranění maximálního prvku (z maximové haldy)

$\mathtt{extractMaxH} : \mathtt{Heap} \dashrightarrow \mathtt{Heap}$

(složitost $\Theta(\log n)$)

Přidání prvku

$\texttt{insertH} : \texttt{Elem} \to \texttt{Heap} \to \texttt{Heap}$

(složitost $\Theta(\log n)$)

Odstranění prvku

$\texttt{removeH} : \texttt{Heap} \to \texttt{Heap} \dashrightarrow \texttt{Heap}$

(složitost $\Theta(\log n)$)

Věta: Neprázdná binární halda na n uzlech má hloubku $\lfloor \log_2 n \rfloor$.

Důkaz indukcí podle n.

Def: Binární strom, v jehož každém podstromu se počet uzlů levého a pravého podstromu liší nejvýše o jednu, se nazývá *uzlově vyvážený* binární strom.

Věta: Každý uzlově vyvážený binární strom je haldou (až na ohodnocení uzlů).

Důkaz: Je třeba dokázat, že délky všech větví uzlově vyváženého stromu se liší nejvýše o jednu.
Předpokládejme sporem, že tvrzení neplatí, a T je nejmenší strom, který tvrzení porušuje. Tedy v každém podstromu stromu T se počet uzlů vlevo a vpravo liší nejvýše o jednu, ale existují zde dvě větve délek m a k, přičemž $m \geq k + 2$. Z minimality stromu T plyne, že jedna z těchto dvou větví, řekněme ta delší, délky m, leží v levém podstromu T_l stromu T, a druhá, kratší, délky k, v pravém podstromu T_r stromu T. Navíc, rovněž z minimality, všechny větve jdoucí do T_l mají délku m nebo $m - 1$, všechny větve jdoucí do T_r mají délku k nebo $k + 1$. To znamená, že T_l má aspoň o dva uzly víc než T_r. Ale podle předpokladu věty má nejvýše o jeden uzel více, což je spor.

Algoritmus řazení binární haldou (funkcionání verze)

```
data Heap  =  Empty  |  Node Int Heap Heap

insHeap :: Int → Heap → Heap
insHeap u Empty         =  Node u Empty Empty
insHeap u (Node v p q)  =  if u ≥ v  then  Node v (insHeap u q) p
                                     else  Node u (insHeap v q) p

toHeap :: [Int] → Heap
toHeap s = foldr insHeap Empty s

toList :: Heap → [Int]
toList Empty = []
toList (Node x l r) = x : merge (toList l) (toList r)

heapSort :: [Int] → [Int]
heapSort  =  toList · toHeap
```

Korektnost řazení haldou

Věta: Algoritmus `heapSort` je totálně korektní řadicí algoritmus.

Konvergence algoritmu je zřejmá. Důkaz parciální korektnosti vyplývá z následujících lemmat.

Lemma: Funkce `toList` vytvoří z haldy h seznam s týmiž prvky jako měla halda h, ale uspořádaný vzestupně.

Lemma: Je-li u číslo a h uzlově vyvážená halda, pak `insHeap` u h je halda obsahující právě prvky z haldy h a prvek u.

Lemma: Funkce `toHeap` vytvoří ze seznamu s haldu obsahující tytéž prvky jako seznam s.

Důkaz prvního lemmatu indukcí podle velikosti haldy.

Důkaz druhého lemmatu: indukcí podle počtu prvků v h se ukáže, že binární strom `insHeap` u h bude opět uzlově vyvážený.
Podle věty o binárních stromech vyvážených vzhledem k počtu uzlů bude výsledek opět halda.
Uspořádání hodnot podél větví vyplývá z definice funkce `insHeap`.

Třetí lemma indukcí podle délky seznamu a z definice kombinátoru `foldr`.

$$\texttt{foldr insHeap Empty } [x_1, x_2, \ldots, x_n]$$
$$= \texttt{insHeap } x_1 \ (\texttt{insHeap } x_2 \ (\ldots(\texttt{insHeap } x_n \texttt{ Empty})\ldots))$$

Složitost řazení haldou

Lemma: Procedura `insHeap` má složitost v $O(\log n)$.

Důkaz: Funkce `insHeap` se rekursivně vyhodnotí právě tolikrát, jako je hloubka haldy. Ale ta je logaritmická vzhledem k počtu jejích uzlů, jichž je vždy nejvýše n.

Věta: Algoritmus řazení haldou má složitost v $\Theta(n \log n)$.

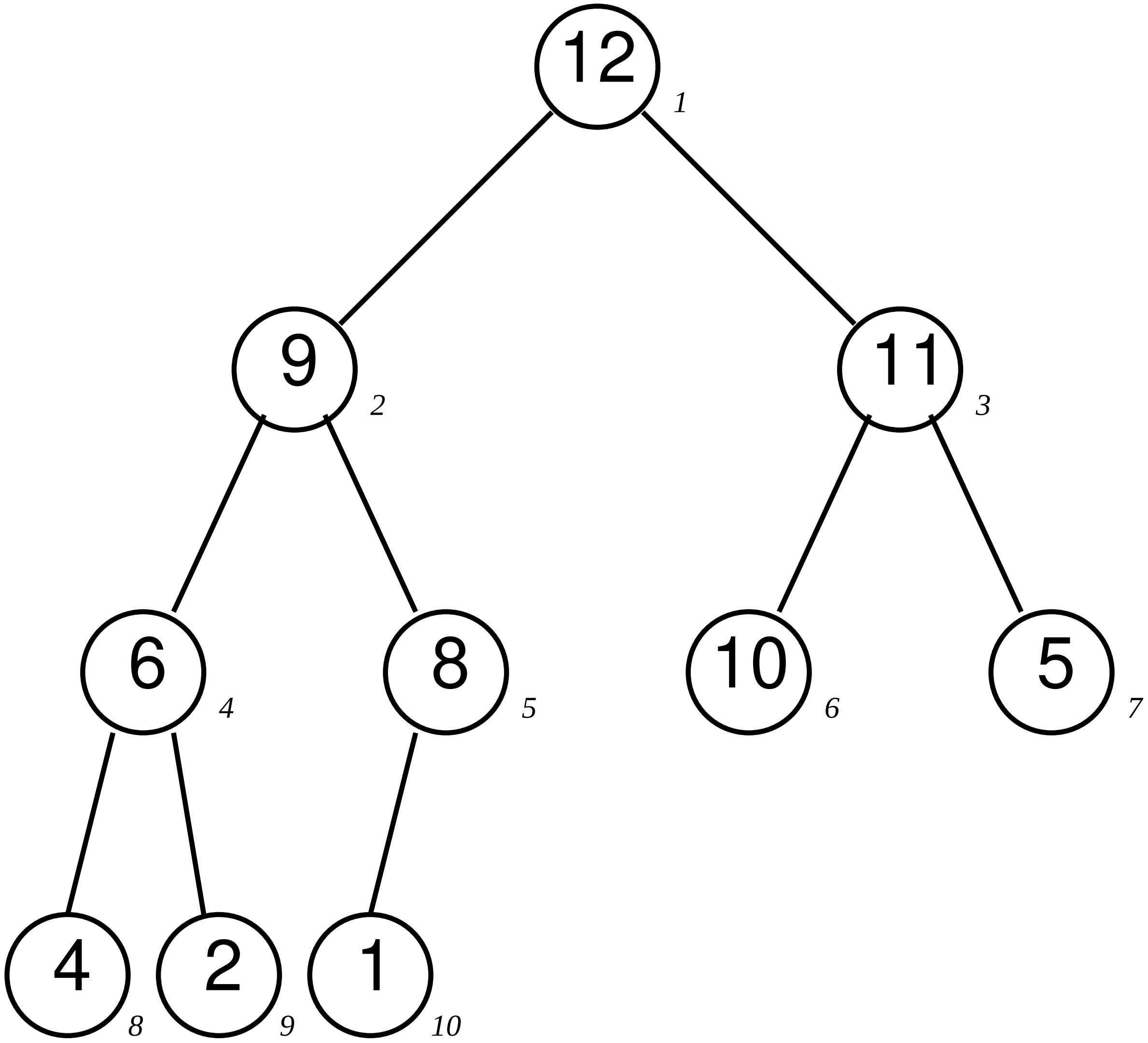
Důkaz: Z předchozího lemmatu vyplývá, že složitost funkce `toHeap` je v $O(n \log n)$. Ve funkci `toList` je hloubka rekursivního zanoření logaritmická, protože seznamy se půlí. Složitost pomocné funkce `merge` je lineární a úhrnná délka výpočtu všech volání funkce `merge` ve stejné hloubce k (tj. se stejně dlouhými seznamy) je $\frac{n}{2^k} \cdot 2^k = n$. Funkce `toList` má tedy složitost také v $O(n \log n)$. Celkem je tedy horní odhad složitosti funkce `heapSort` $n \log n$, ale podle Věty o dolním odhadu složitosti řadicích je toto i dolním odhadem.

V imperativním algoritmu `heapSort` se používá maximová halda, tj. binární halda, která má položky ve větvích seřazeny *sestupně*. Tedy největší položka je vždy v kořeni.

Jelikož algoritmus má haldu efektivně reprezentovánu polem, je zde výhodnější uvažovat binární haldu nikoliv uzlově vyváženou, ale naopak s levým podstromem „těžším“.

Def: *Vlevo zarovnaná halda* je binární halda, jejíž všechny větve délky k leží nalevo od větví délky $k-1$.

Vlevo zarovnaná halda



Věta: Reprezentujme vlevo zarovnanou binární haldu na n uzlech posloupností hodnot jejích uzlů $(a_1, \ldots, a_n)$ takto: a_1 reprezentuje kořen haldy a je-li uzel u reprezentován prvkem a_i, pak levý následník uzlu u je reprezentován prvkem a_{2i} a pravý následník uzlu u je reprezentován prvkem a_{2i+1}. Pak posloupnost $(a_1, \ldots, a_n)$ je haldou určena jednoznačně.

Věta umožňuje pracovat s vlevo zarovnanou binární haldou reprezentovanou v paměti polem. Vlevo zarovnaná binární halda je totéž jako pole rozepsané „po vrstvách“.

Algoritmus řazení binární haldou (imp. verze)

```
type  Element = Int;
      Posl = array [1..MAX] of Element;

procedure heapSort (n:Int, var A:Posl);
  var l,r: Int;  x: Element;

  procedure createHeap (l,r:Int);
    { vytvoření haldy A[l]...A[r] ze dvou podhald začínajících A[2*l] a A[2*l+1] }
    var i,j:Int; {pom. indexy}  y:Element; {zařazovaný prvek}  p:Bool; {sestupovat?}
    begin i := l;  j := 2*i;  y := A[i];  p := True;
          while  p  &&  (j≤r) do  { sestupujeme }
            begin if (j<r) && (A[j]<A[j+1]) then j:=j+1;
                  { A[j] je teď větší z kořenů podhald pod A[i] }
                  p := y<A[j];
                  if p then { A[j]>y, takže posuneme A[j] o patro výš }
                            begin
                              A[i] := A[j];
                              i := j;  j := 2*i
                            end
            end;
          A[i] := y  { prvek y zařazen na své místo }
    end {createHeap};
```

```
begin {heapSort}
   { postupně vytvoříme haldu A[1],...,A[n] }
        for l := n div 2 downto 1 do  createHeap(l,n);

   { V kořenu (A[1]) je prvek s největším ohodnocením.}
   { Vyměníme ho s koncem pole, haldu zkrátíme        }
   { a restaurujeme zařazením A[1] na správné místo   }
        for r := n downto 2 do
         begin
            x := A[1];  A[1] := A[r];  A[r] := x;
            createHeap (1,r-1)
         end
end {heapSort};
```

Korektnost a složitost imperativního algoritmu `heapSort`

Věta: Mějme vlevo zarovnanou binární haldu reprezentovanou posloupností $(a_l, \ldots, a_r)$ a nechť oba podstromy pod kořenem splňují podmínky haldy. Pak posloupnost $(a'_l, \ldots, a'_r)$ ve stavu po provedení $\texttt{createHeap}(a, l, r)$ splňuje podmínky (vlevo zarovnané) binární haldy.

Důkaz indukcí podle hloubky haldy.

Věta: Provedení procedury $\texttt{heapSort}(A)$ seřadí posloupnost A

Idea důkazu: Jednoprvkové podstromy reprezentované druhou polovinou posloupnosti A jsou triviální podhaldy. První fáze algoritmu vytváří z těchto podhald větší podhaldy. Na konci první fáze je posloupnost A binární haldou.

Druhá fáze algoritmu vytvoří z haldy seřazenou posloupnost. To vyplývá z toho, že největší prvek haldy je vždy v jejím kořenu: největší prvky se skládají na konec posloupnosti.

Věta: Algoritmus řazení haldou konverguje.

Důkaz: Tvrzení věty vyplývá z konečné velikosti a hloubky haldy.

Věta: Řazení haldou (imperativní algoritmus) má složitost v $\Theta(n \log n)$.

Důkaz: První fáze (vytváření haldy) má složitost $O\left(\frac{n}{2} \log n\right)$, druhá fáze má složitost $O(n \log n)$, obě tedy dohromady $O(n \log n)$.

Podle Věty o dolním odhadu složitosti musí být rovněž v $\Omega(n \log n)$.

Z obou odhadů pak plyne tvrzení.