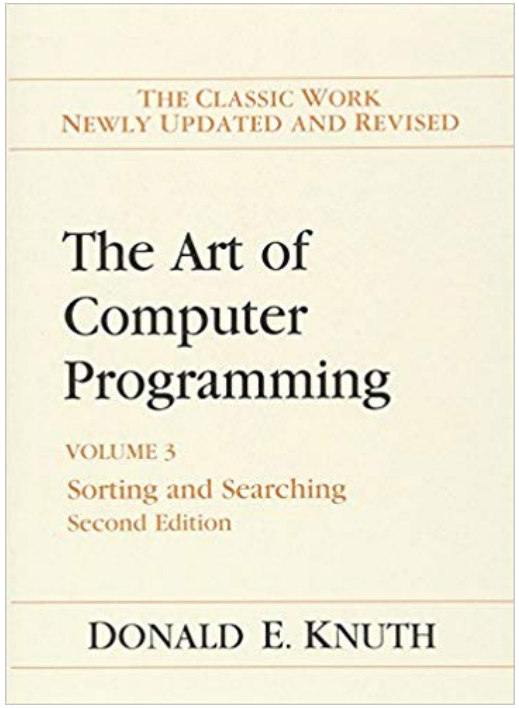
Algoritmizace

Třídění



Vnitřní třídění

Vstup: pole `a` s prvky, které lze porovnávat

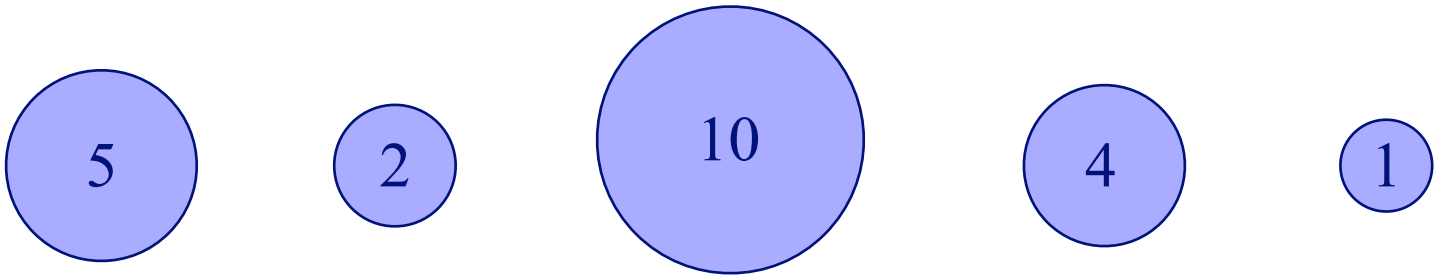
Výstup: pole `a` s prvky uspořádanými vzestupně

Bublinkové třídění BubbleSort

- projdi pole a porovnej dvojice sousedních prvků
- v případě potřeby dvojici vyměň
- po dosažení konce seznamu začni znovu od začátku
- po i-té iteraci ($i = 1, 2, \ldots, n\text{-}1$) je i posledních prvků na svých místech, není je již tedy třeba porovnávat
- po n-1 iteracích (`n=len(a)`) výpočet končí

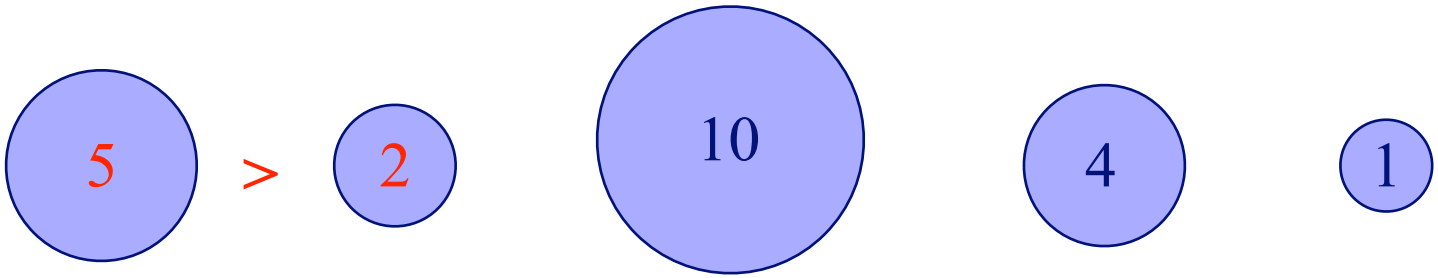
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
        if a[j] > a[j+1]:
            a[j],a[j+1] = a[j+1],a[j]
```



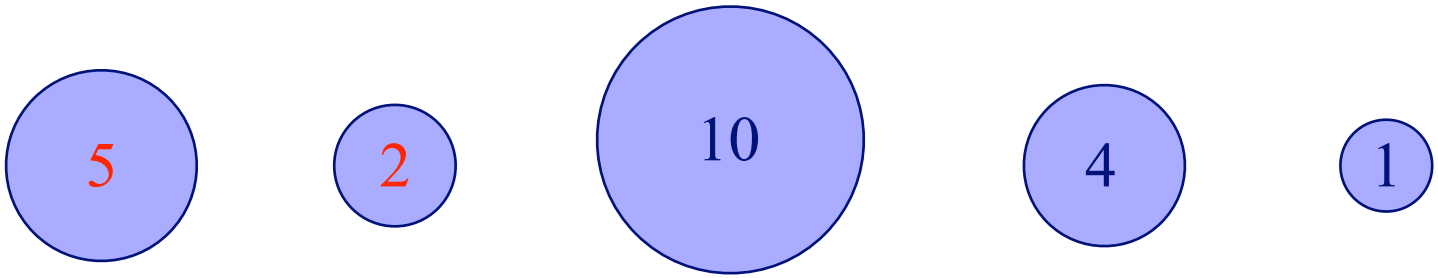
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



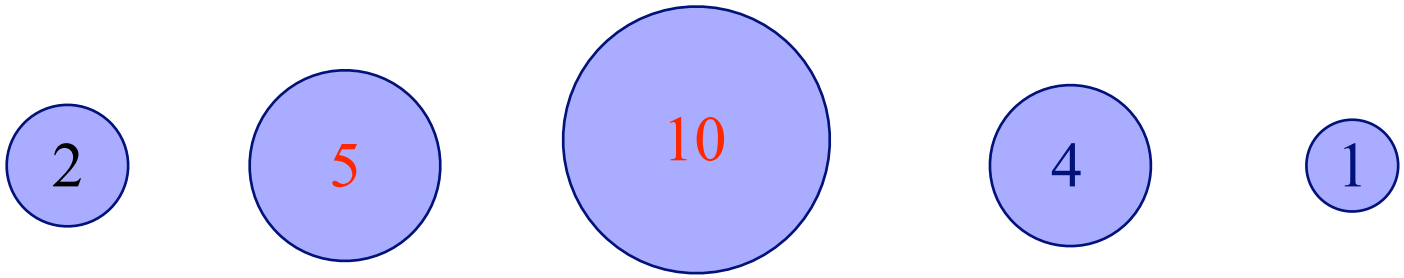
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



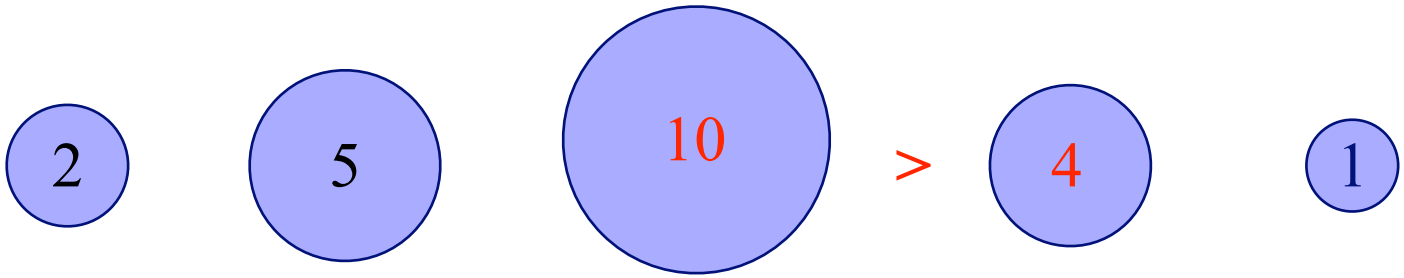
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



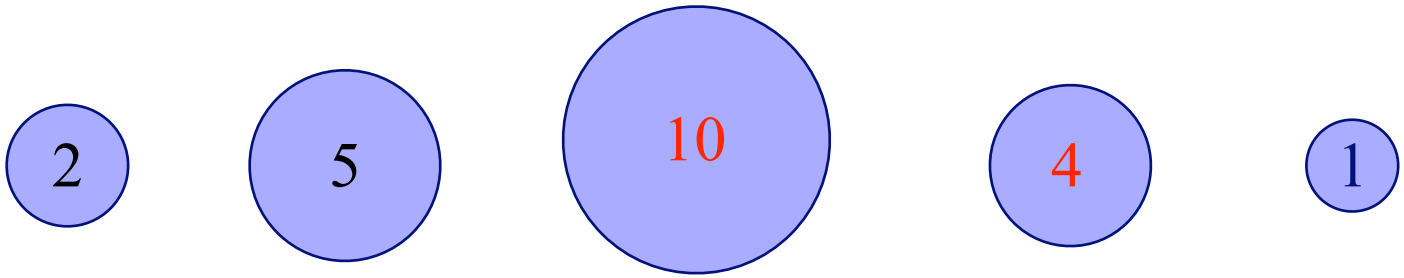
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



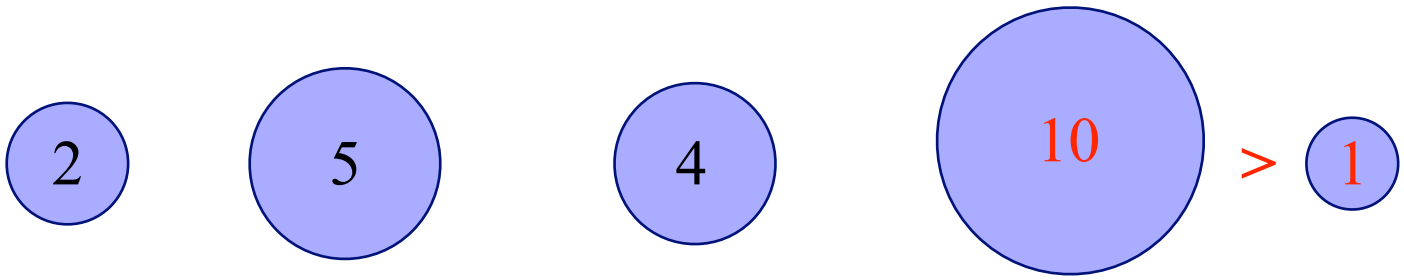
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



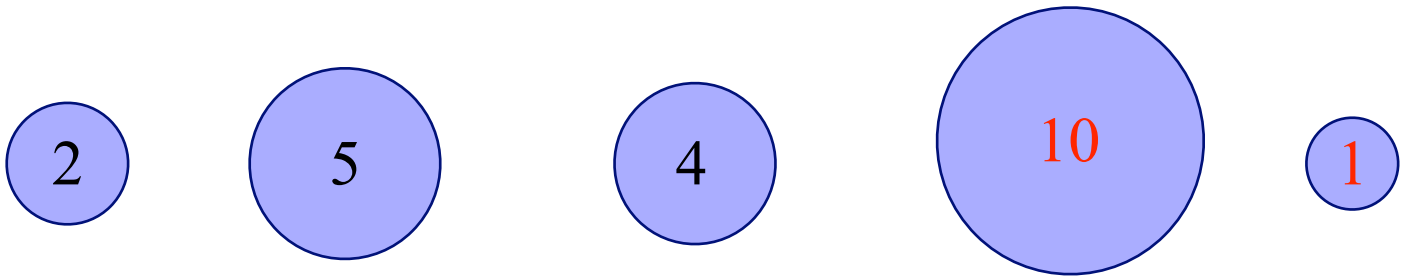
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



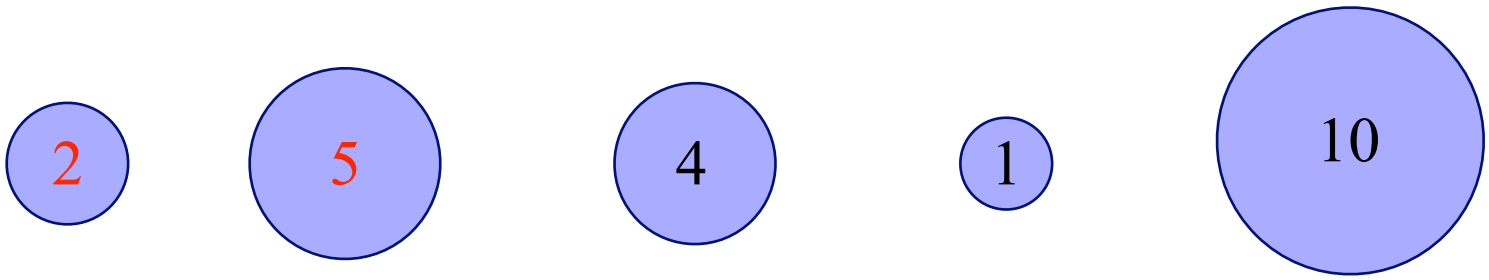
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



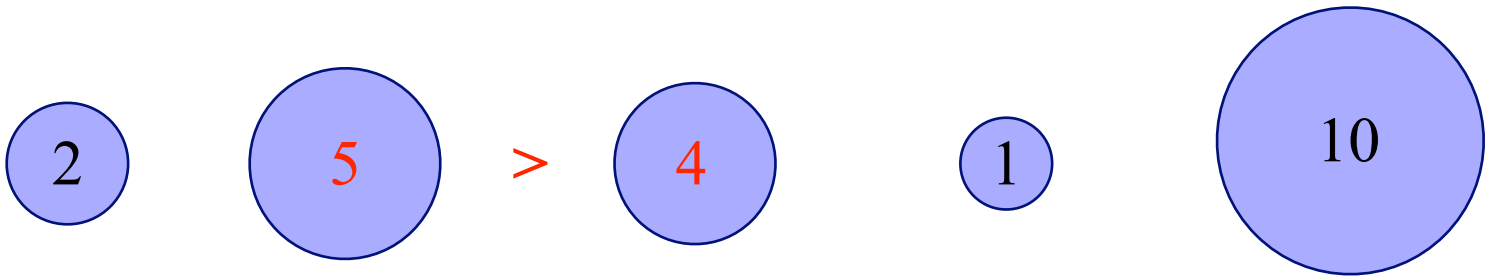
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



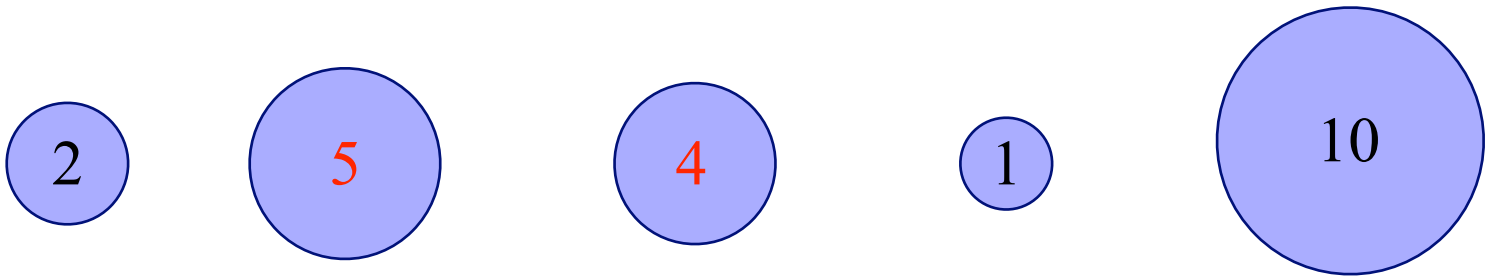
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



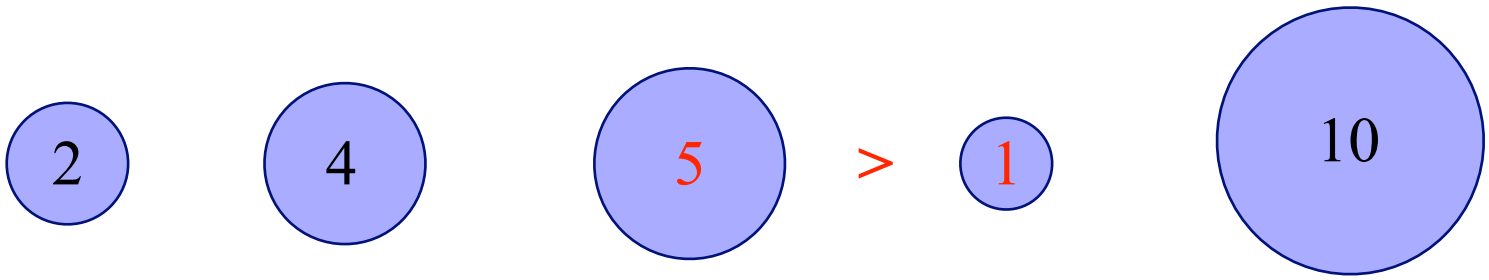
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



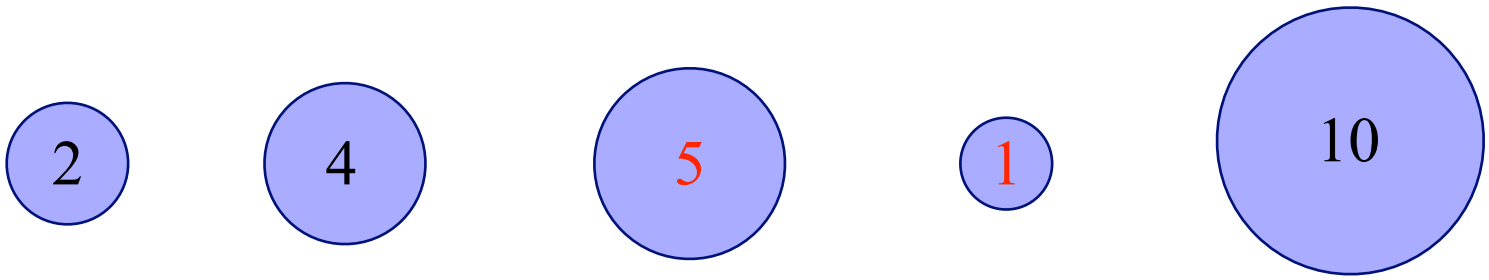
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
        if a[j] > a[j+1]:
            a[j],a[j+1] = a[j+1],a[j]
```



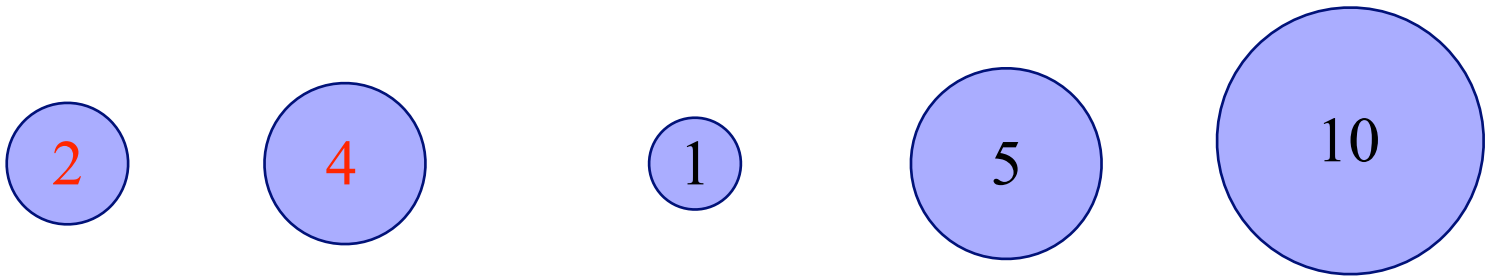
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



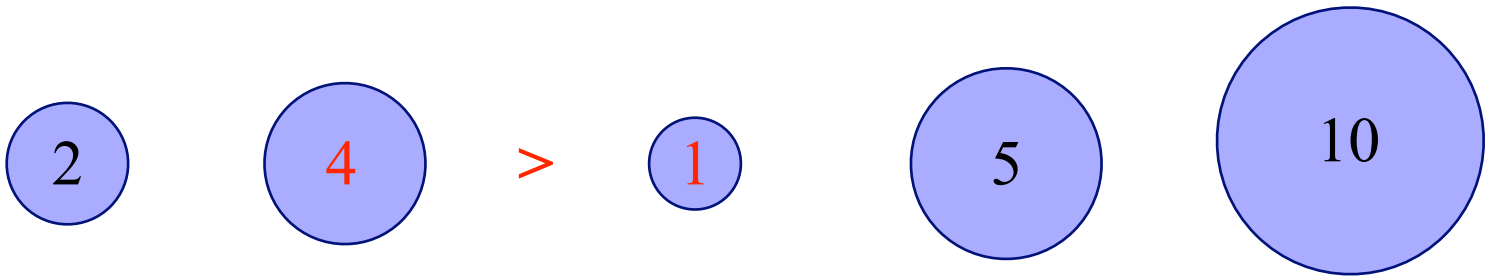
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



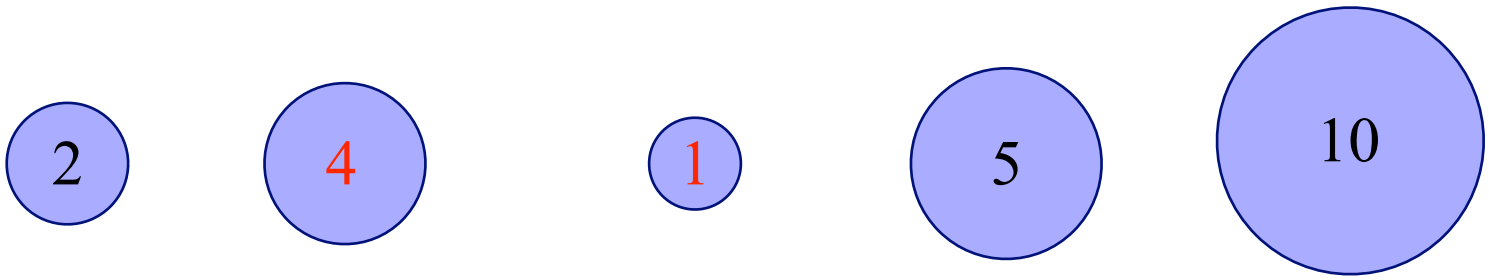
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



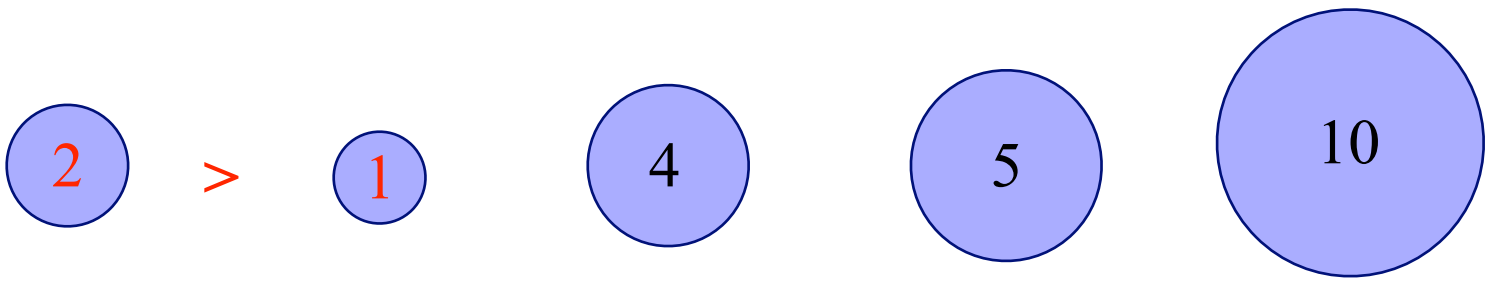
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
        if a[j] > a[j+1]:
            a[j],a[j+1] = a[j+1],a[j]
```



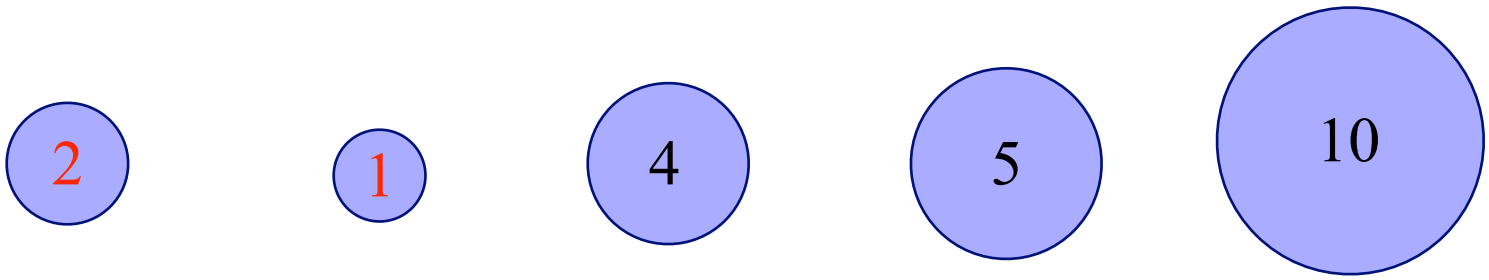
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
        if a[j] > a[j+1]:
            a[j],a[j+1] = a[j+1],a[j]
```



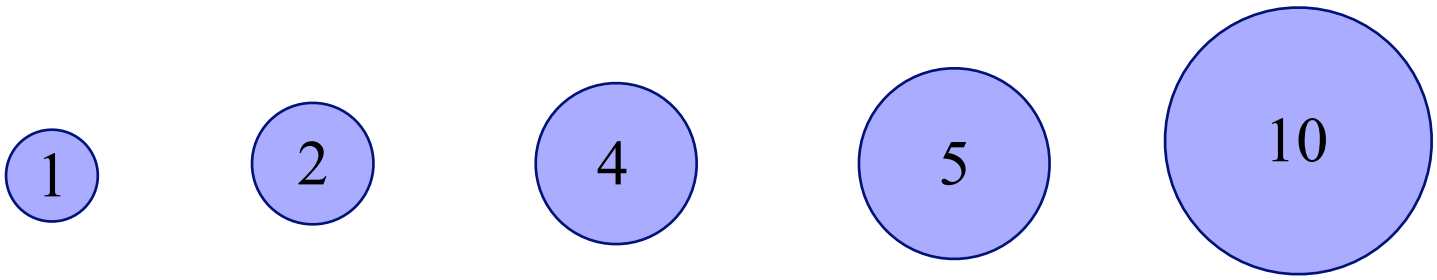
Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



Příklad

```
for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
    for j in range(i):    # 0 ≤ j ≤ i-1
      if a[j] > a[j+1]:
          a[j],a[j+1] = a[j+1],a[j]
```



Bublinkové třídění BubbleSort

```
def bubbleSort(a):
    n = len(a)
    for i in range(n-1,0,-1): # n-1 ≥ i ≥ 1
        # prvky a[i+1:n]
        # jsou již na svých místech
        for j in range(i):        # 0 ≤ j ≤ i-1
            # a[j] = max a[0..j]   0 ≤ j ≤ i
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
        # a[i] = max a[0..i]
```

invariant

invariant

- Časová složitost $\Theta(n^2)$

✎ Problém

① Zlepšení algoritmu bubbleSort

- jedna iterace převede maximální prvek na poslední místo mezi prvky, které jsou předmětem porovnání
- pokud poslední výměna byla mezi a[j] ↔ a[j+1], znamená to, že a[j+1] ≤ a[j+2] ≤ a[j+3] ≤ …
- v další iteraci tedy netřeba porovávat a[i] pro i > j

Vylepšete algoritmus tak, aby každá iterace skončila na pozici poslední výměny v předchozím kroku. Změní to nějak časovou složitost v nejhorším případě?

Třídění výběrem SelectionSort

První krok

- najdi minimální prvek
- a vyměň s prvkem na pozici 0

Další krok

- mezi zbývajícími prvky najdi minimální
- a vyměň s prvkem na pozici 1

Obecný krok (pro `i` = `0,1,…,n-2`)

- mezi `a[i],…,a[n-1]` najdi minimální prvek
- a vyměň s prvkem `a[i]`

Invariant cyklu

- na začátku každé iterace (pro `i` = `0,1,…,n-1`) platí
- prvních `i` prvků tvoří setříděný úsek
- který obsahuje `i` minimálních prvků pole `a`

Třídění výběrem SelectionSort

```
def selectionSort(a):
    n = len(a)
    for i in range(n-1):
        # a[i] vyměň s minimem z a[i:]
        minIndex, minHodnota = i, a[i]
        # dočasné minimum
        for j in range(i+1,n):
            if minHodnota > a[j]:
                minHodnota = a[j]
                minIndex = j
        a[i], a[minIndex] = a[minIndex], a[i]
```

Třídění výběrem – analýza

✖ Proti: Časová složitost $\Theta(n^2)$

Data malého rozsahu (desítky prvků)

- lepší nežli BubbleSort

✔ Pro:

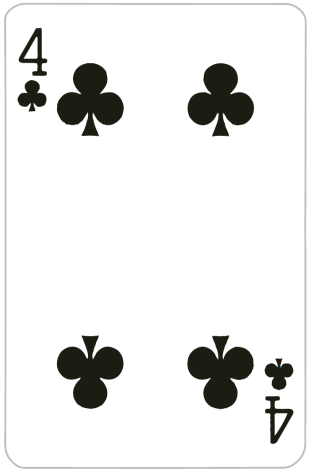
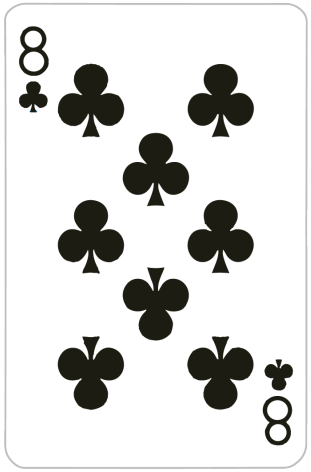
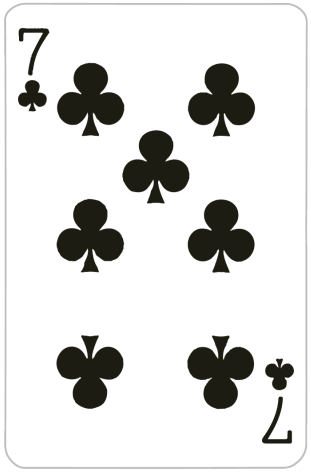
- Jen $n - 1$ výměn
- jen $O(n)$ zápisů do pole `a`

Třídění vkládáním InsertionSort

Jako třídíme karty

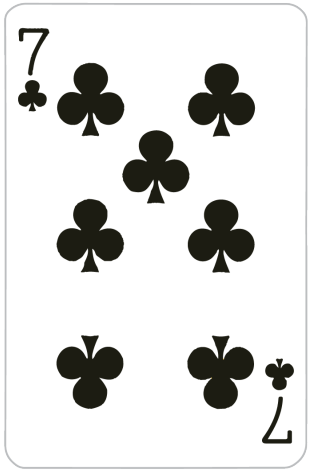
- vezměte novou kartu z balíčku
- a postupným porovnáváním zprava doleva
- s již setříděnými kartami, které držíte v ruce
- vložte na správné místo

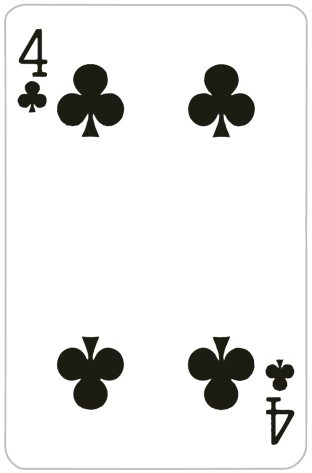
Příklad

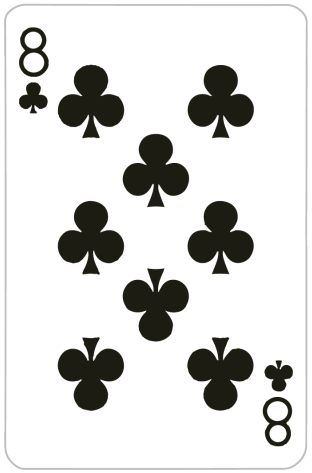




$4 < 8$ $8 > 7$

Příklad

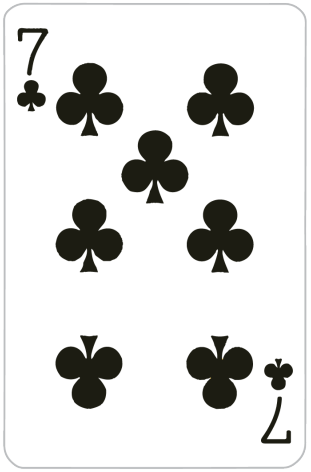


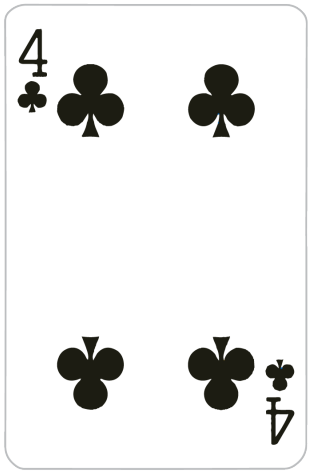




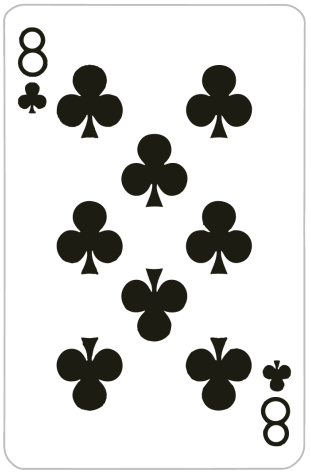
$8 > 7$

Příklad

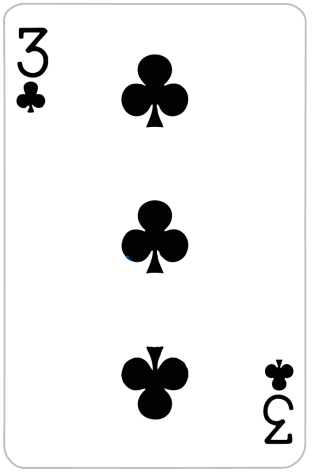


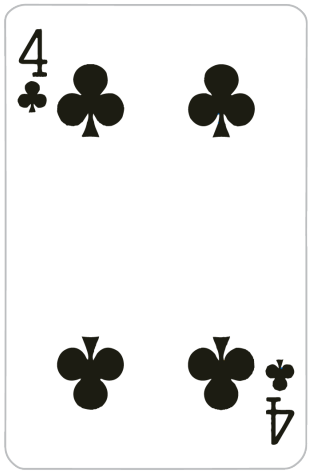


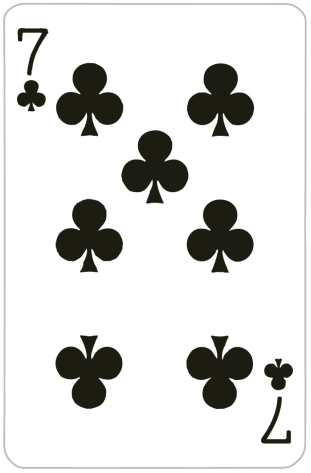
$4 < 7$

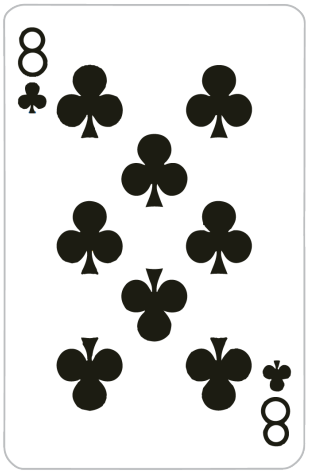


Příklad



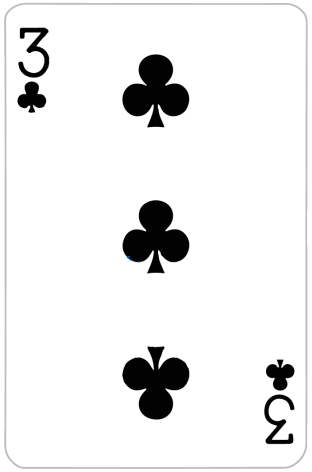


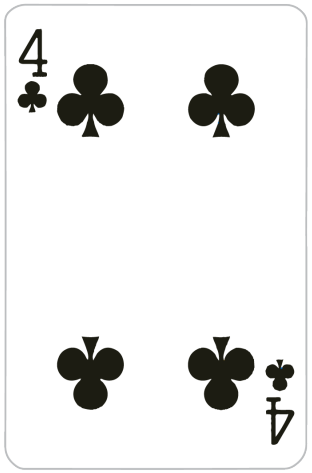


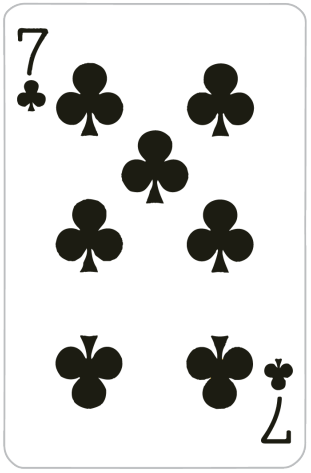


$8 > 3$

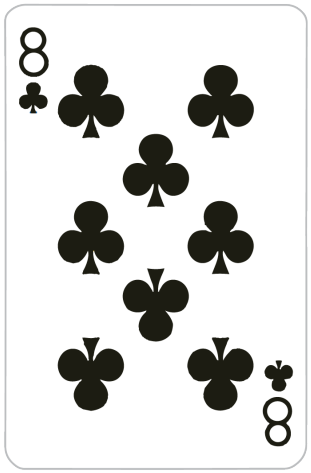
Příklad



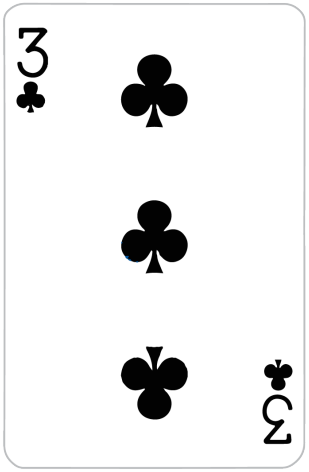


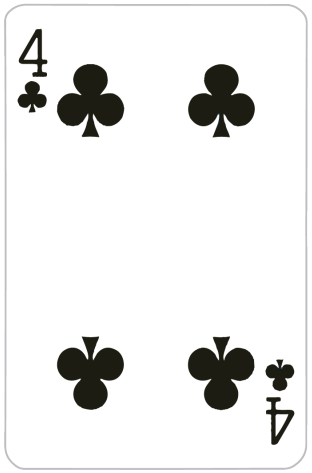


$7 > 3$

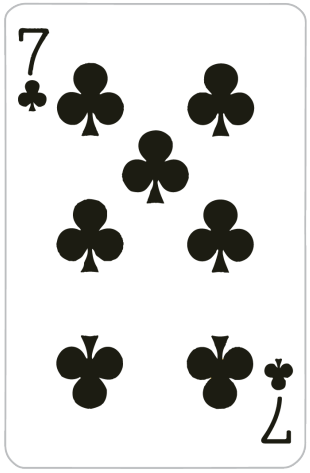


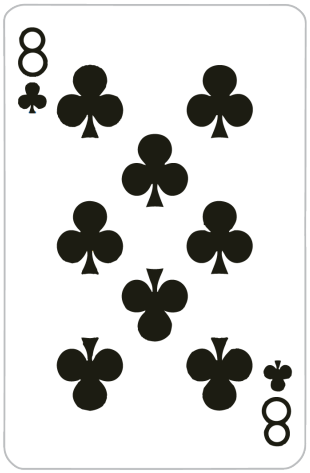
Příklad



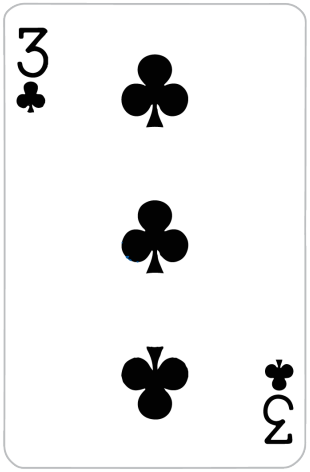


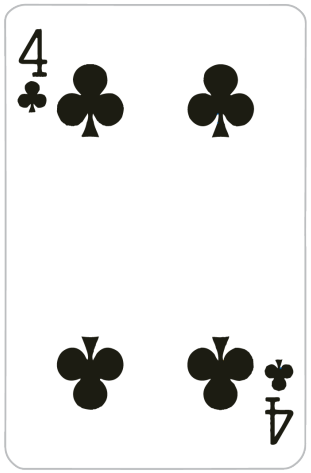
$4 > 3$

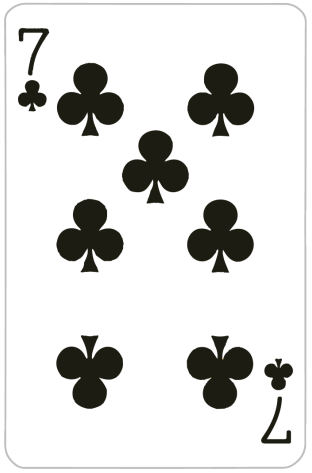


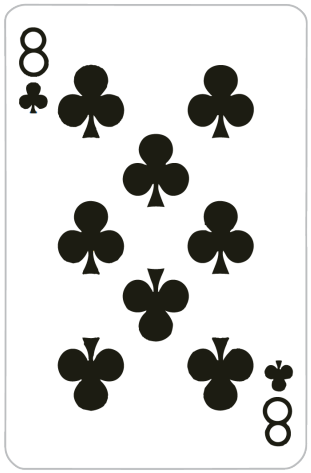


Příklad









Třídění vkládáním InsertionSort

```
def insertionSort(a):
    n = len(a)
    for i in range(1,n):
    # vlož a[i] do setříděného a[:i]
        x, j = a[i], i-1
        while j >= 0 and a[j] > x:
            a[j+1] = a[j]
            j -= 1
        a[j] = x
```

Invariant: na začátku `i`-té iterace (`i` = `1,2,...,n`) je `a[:i]` setříděno

Třídění vkládáním – analýza

✖ Časová složitost $\Theta(n^2)$

✔ Vhodné pro data malého rozsahu (desítky prvků)

- lepší nežli BubbleSort

Srovnání s SelectionSort

- SelectionSort musí vždy projít zbývající prvky pro nalezení minima
- InsertionSort může stačit jen jediné porovnání
- ✔ výhodné pro částečně setříděné vstupy
- ✔ v průměrném případě provede cca polovinu porovnání ve srovnání se SelectionSort

Haldové třídění HeapSort

Datová struktura binární halda (binary heap)

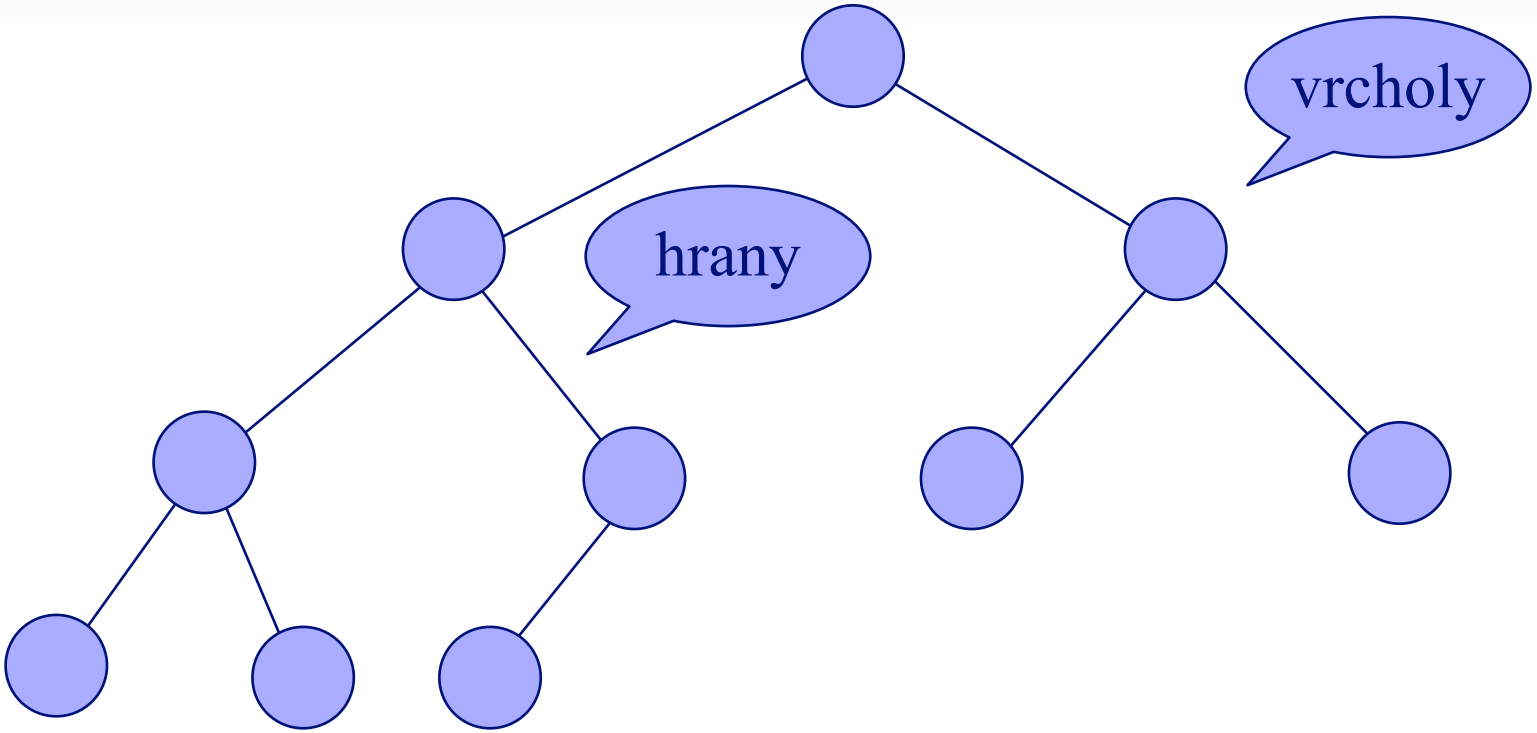
Operace

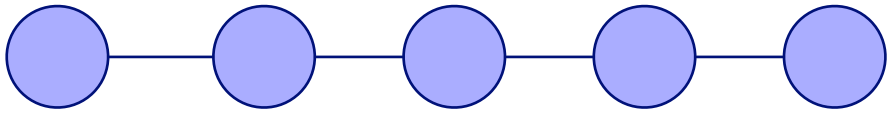
- Přidej - vložení nového prvku
- OdeberMin - odebrání minimálního prvku
- lze provést v čase O(log n)
- n = počet prvků uložených v haldě

HeapSort

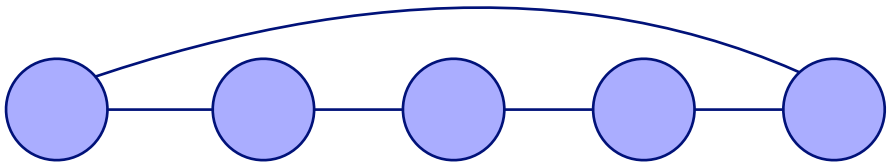
- z n zadaných prvků postav haldu : čas O(n log n)
- n-krát odeber minimum : čas O(n log n)
- třídění v čase O(n log n)

Graf



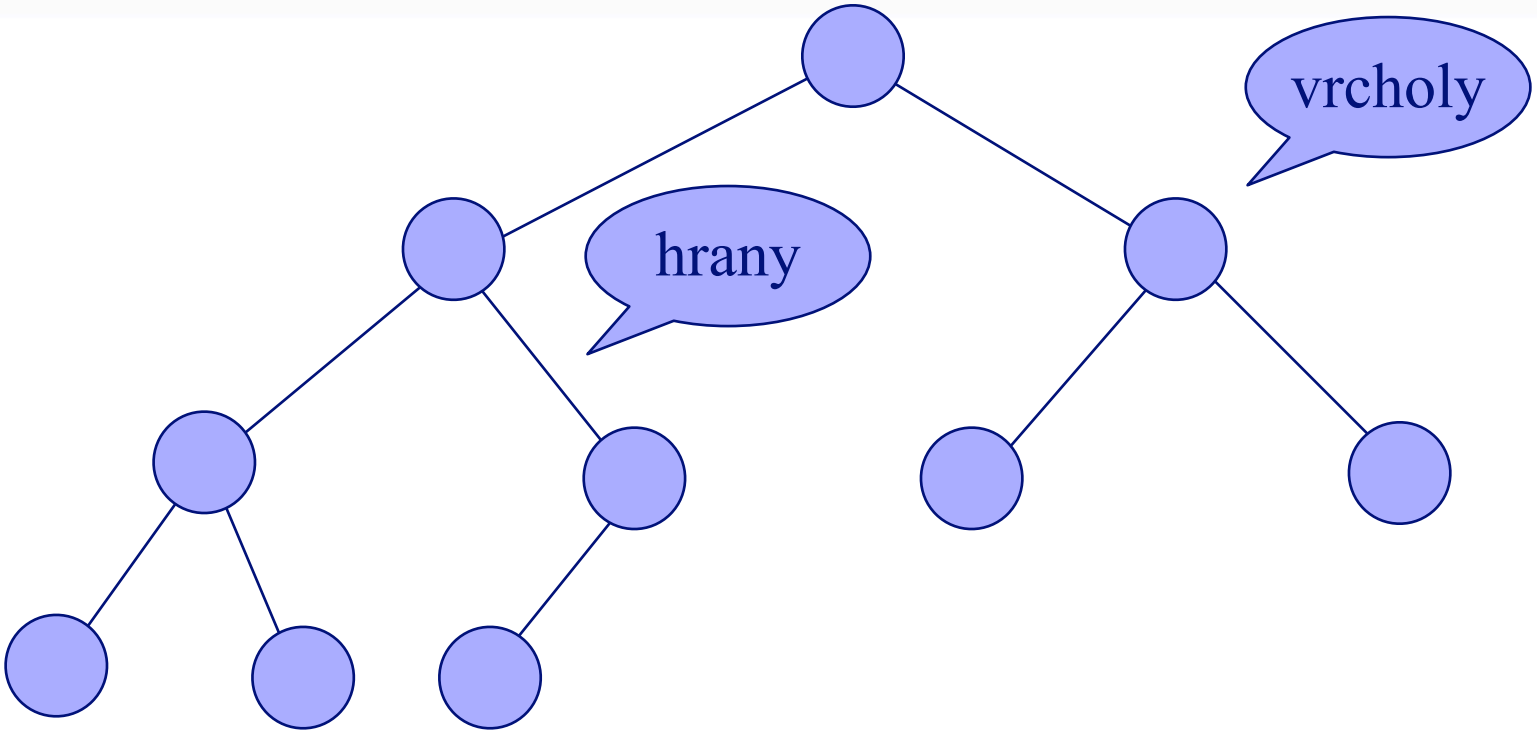


cesta (délky 4)



kružnice (délky 5)

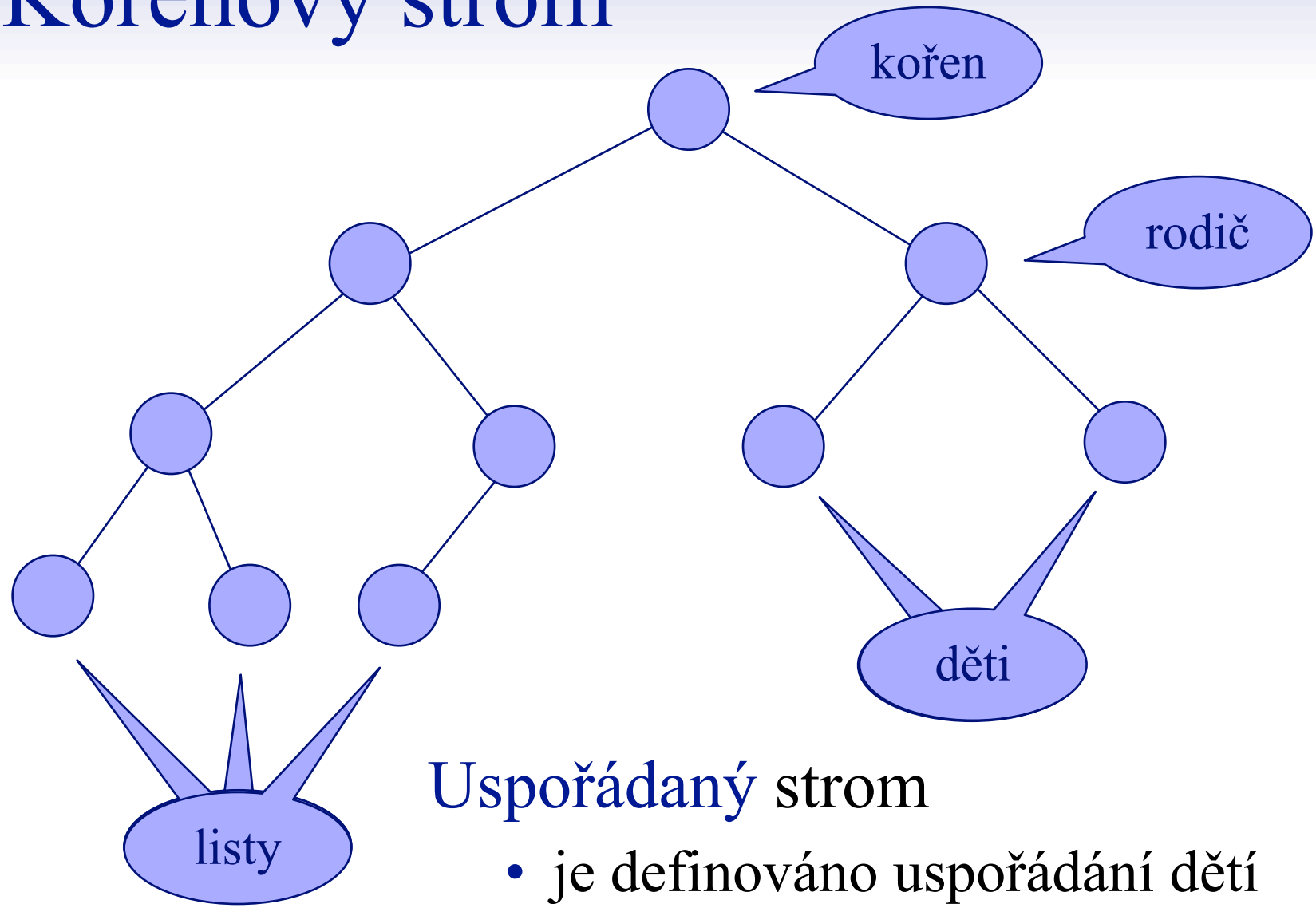
Graf



Graf je

- souvislý - mezi každou dvojící vrcholů existuje cesta
- strom - souvislý a neobsahuje kružnici

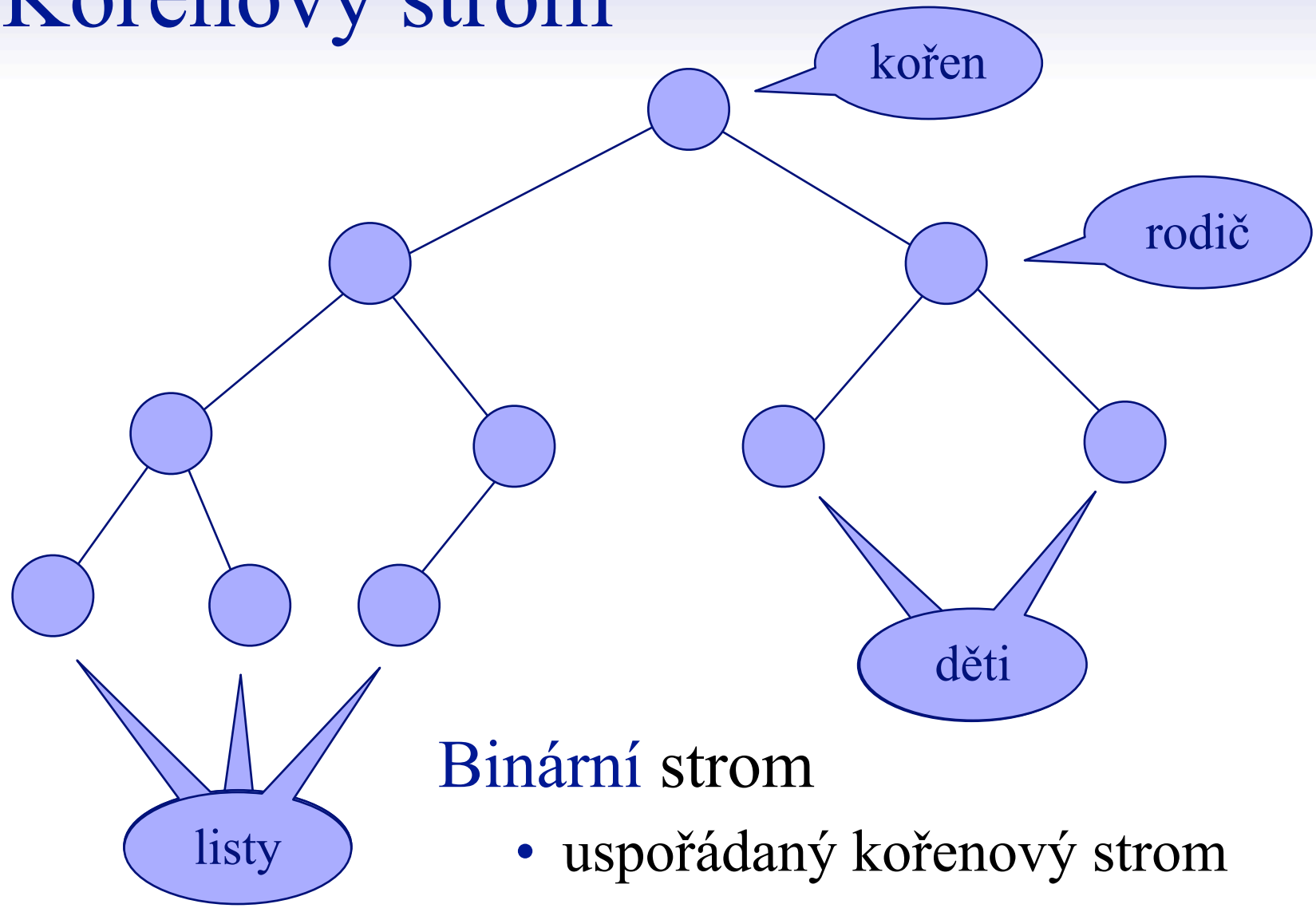
Kořenový strom



Uspořádaný strom

- je definováno uspořádání dětí
- pro každého rodiče

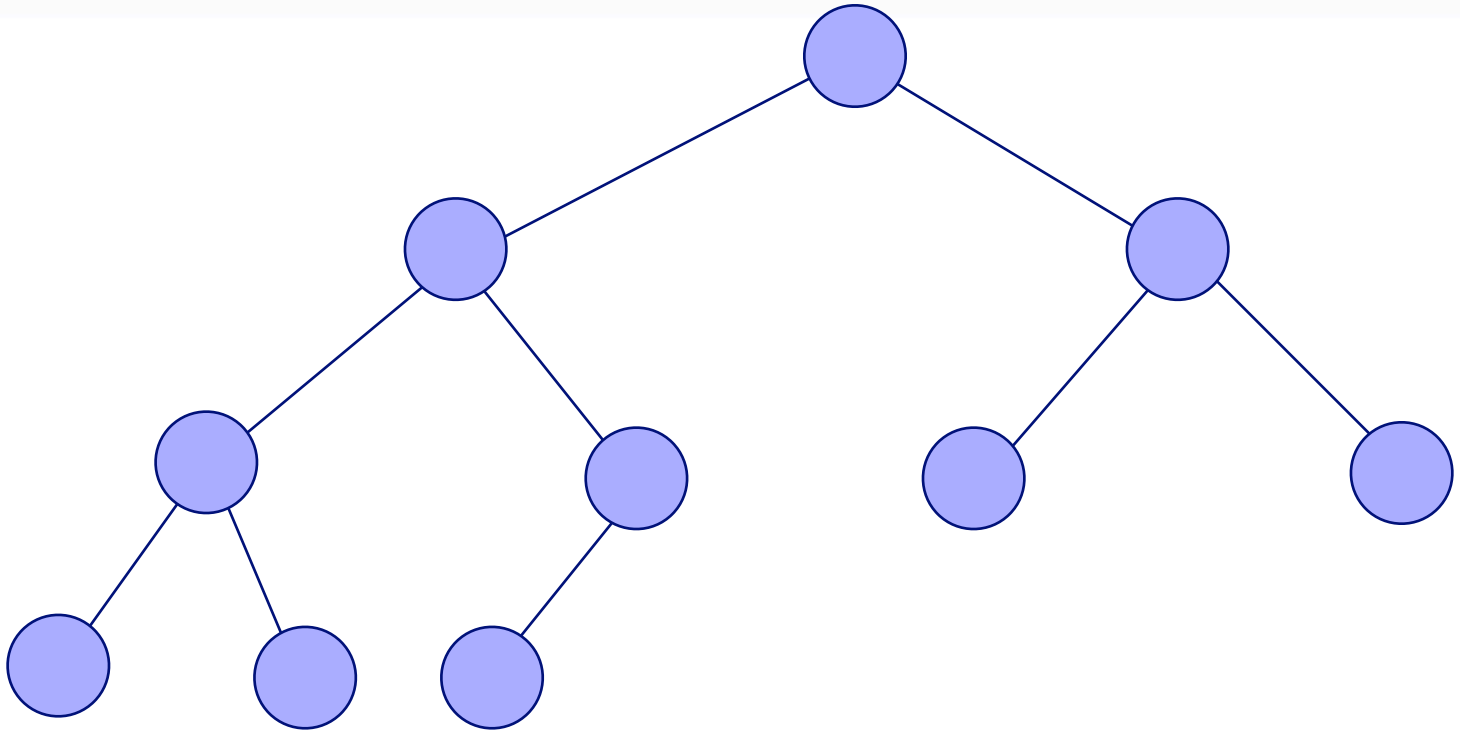
Kořenový strom



Binární strom

- uspořádaný kořenový strom
- každý rodič má nejvýše dvě děti

Kořenový strom



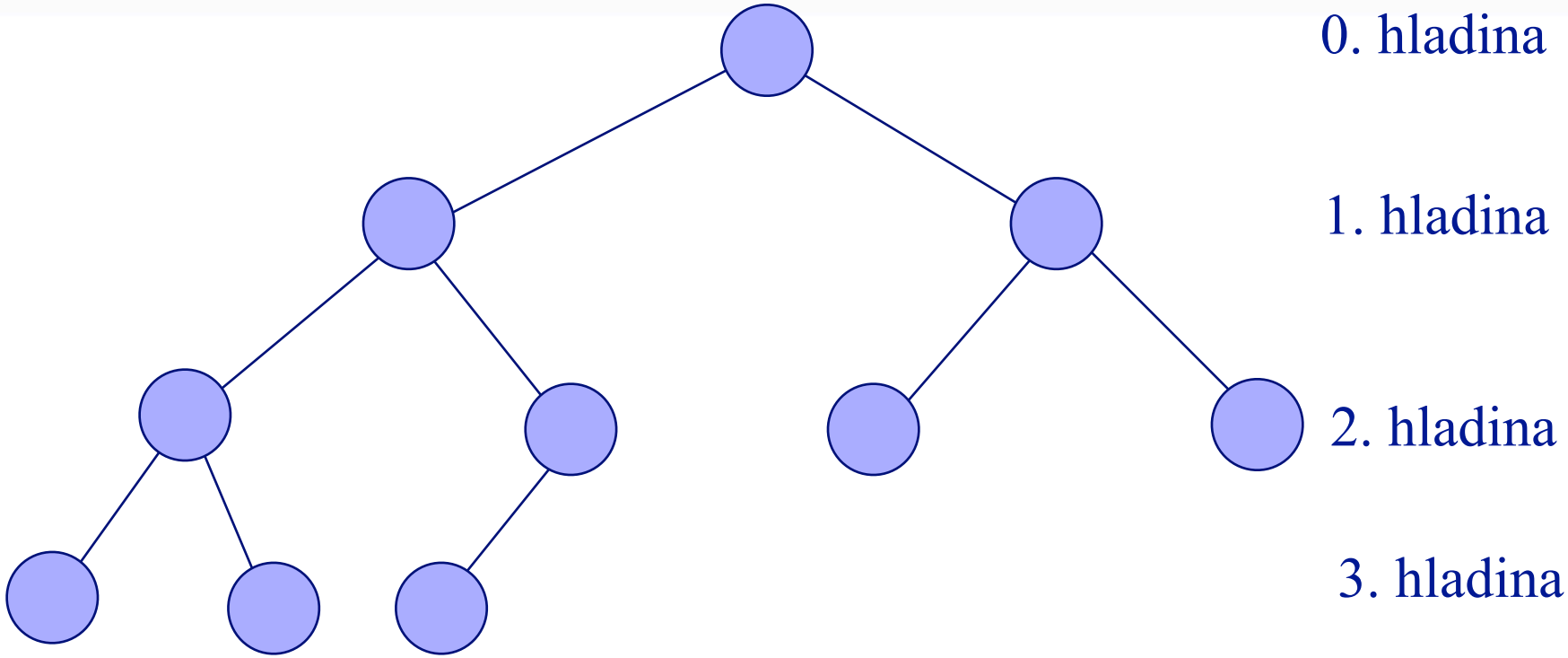
Vzdálenost vrcholů u a v

- délka nejkratší cesty mezi u a v

Výška stromu

- délka nejdelší cesty z kořene do listu

Kořenový strom

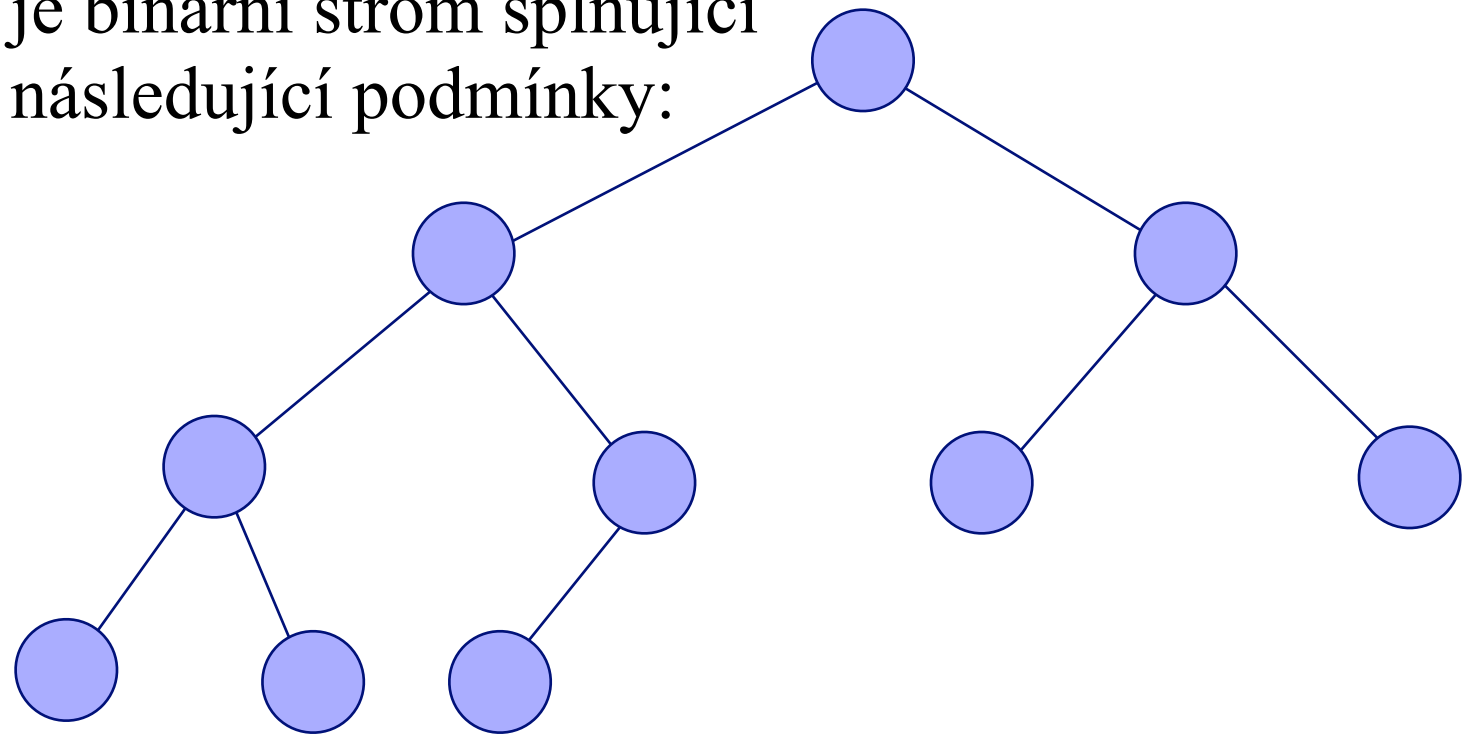


i-tá hladina

- je tvořena vrcholy ve vzdálenosti *i* od kořene

Binární halda

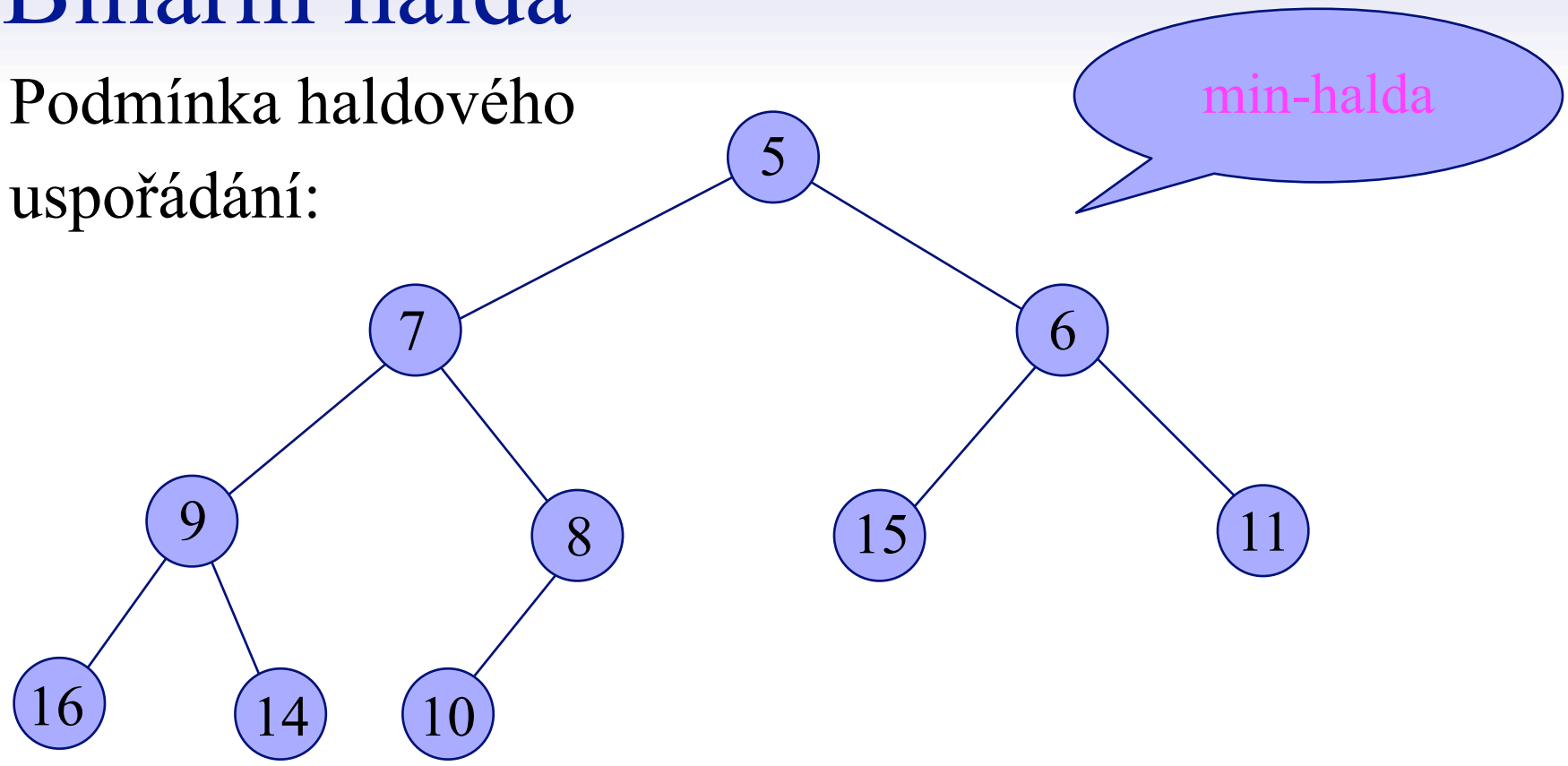
je binární strom splňující následující podmínky:



- v každé hladině od první do předposlední je max # vrcholů
- poslední hladina se zaplňuje zleva
- hodnoty uložené ve vrcholech splňují podmínku haldového uspořádání

Binární halda

Podmínka haldového uspořádání:

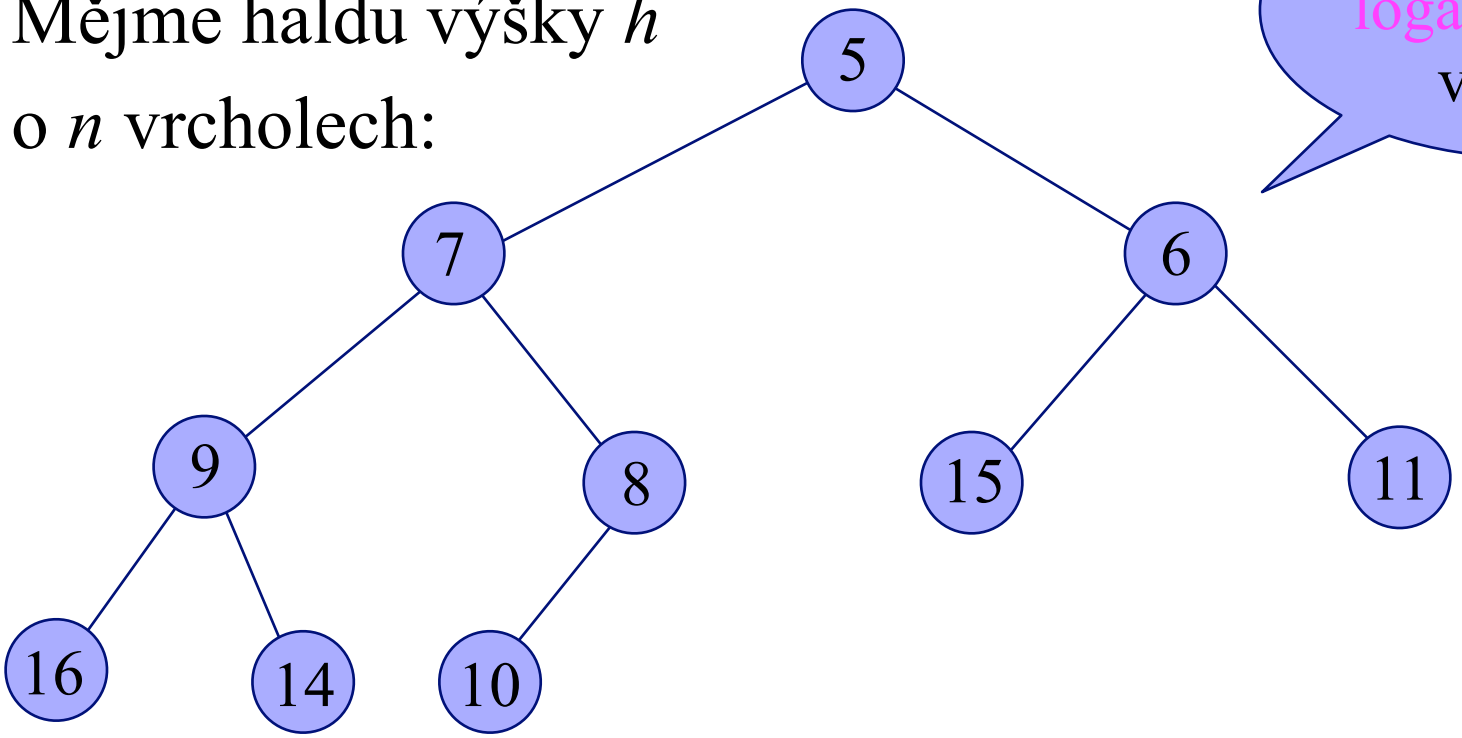


Pro každý vrchol platí, že hodnota v něm uložená je

- menší nebo rovna než hodnota v libovolném z jeho dětí (min-halda)
- větší nebo rovna než hodnota v libovolném z jeho dětí (max-halda)

Binární halda – vlastnosti

Mějme haldu výšky h o n vrcholech:



halda má logaritmickou výšku !

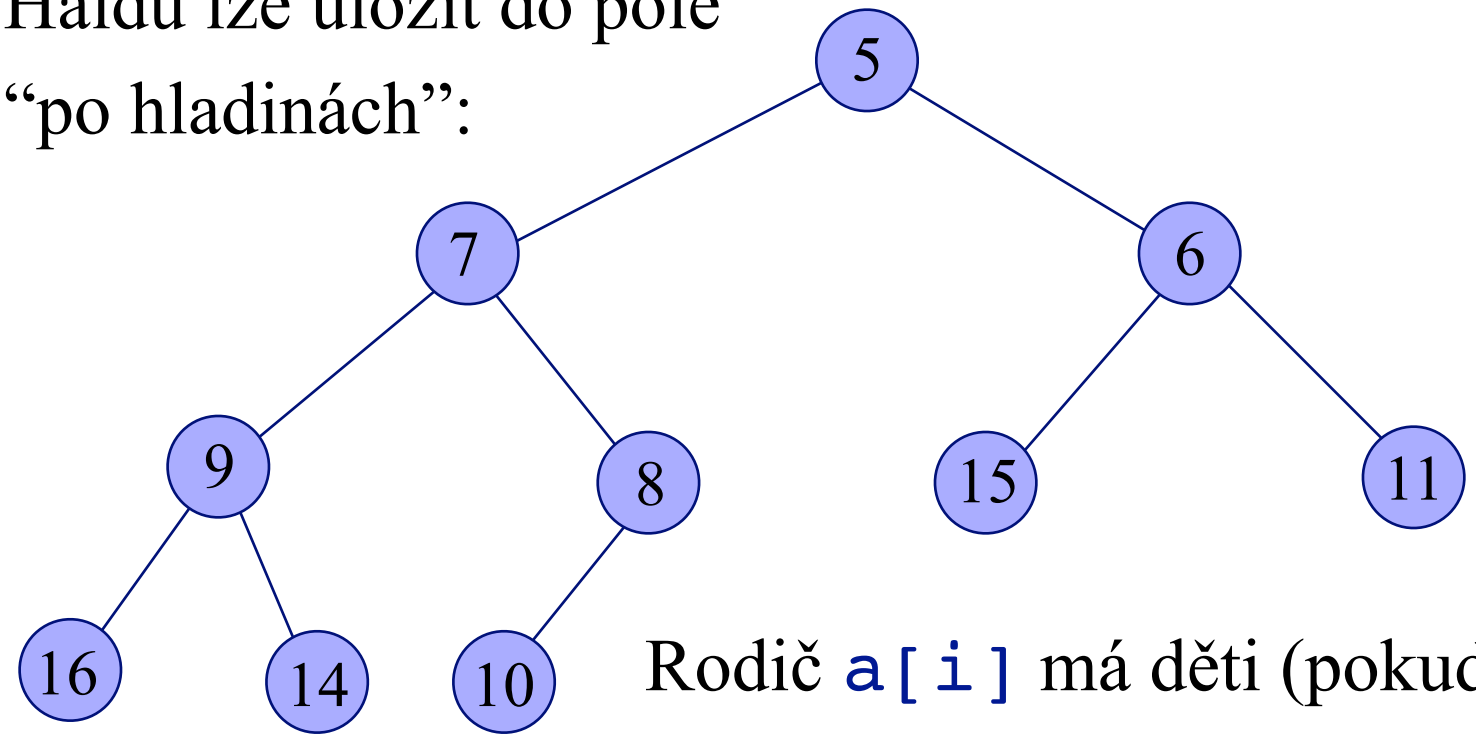
Pak platí

- pro $0 \leq i \leq h-1$ je na i-té hladině 2^i vrcholů
- na poslední hladině je alespoň 1 vrchol

Tedy $n \geq \sum_{i=0}^{h-1} 2^i + 1 = 2^h \implies h \leq \log_2 n$

Binární halda – vlastnosti

Haldu lze uložit do pole “po hladinách”:

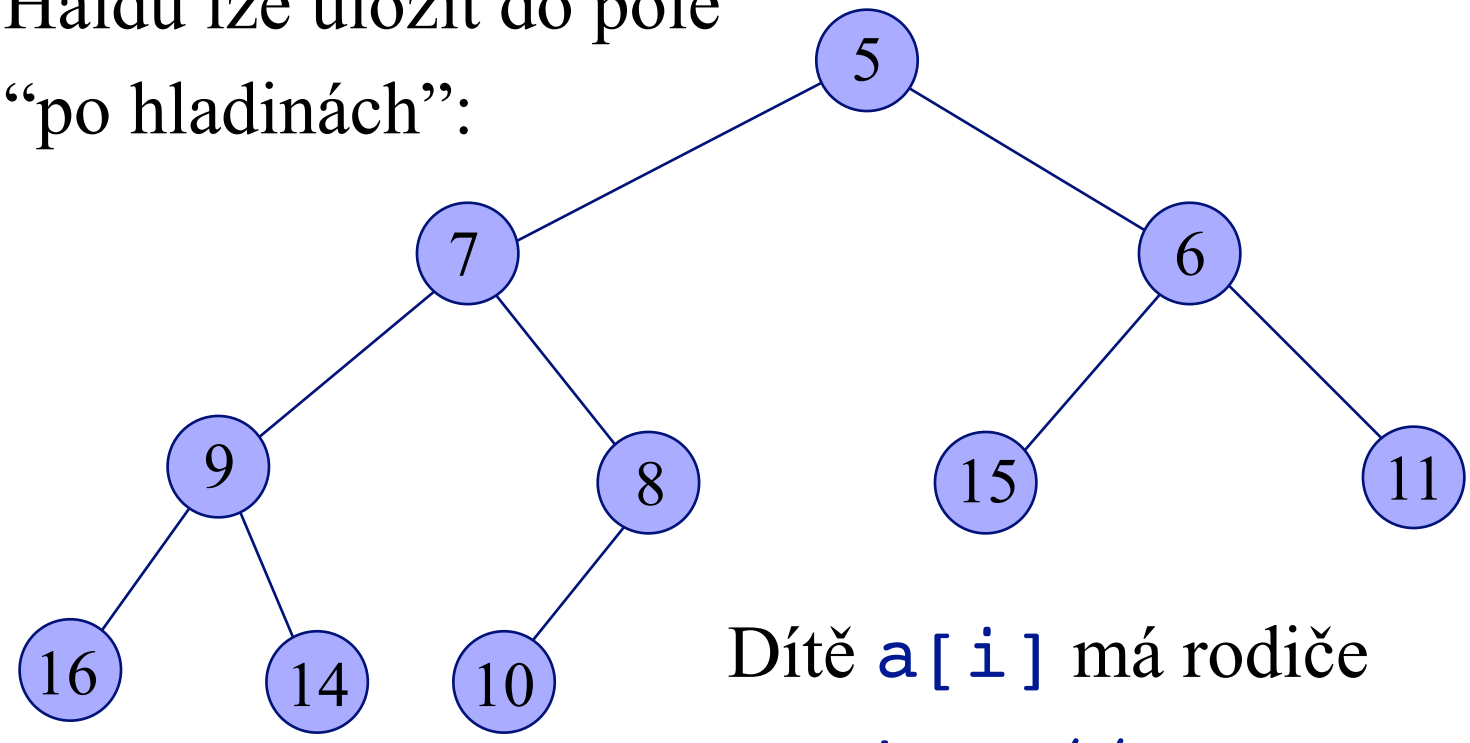


Rodič `a[i]` má děti (pokud existují) `a[2*i+1]` a `a[2*i+2]`

0	1	2	3	4	5	6	7	8	9
5	7	6	9	8	15	11	16	14	10

Binární halda – vlastnosti

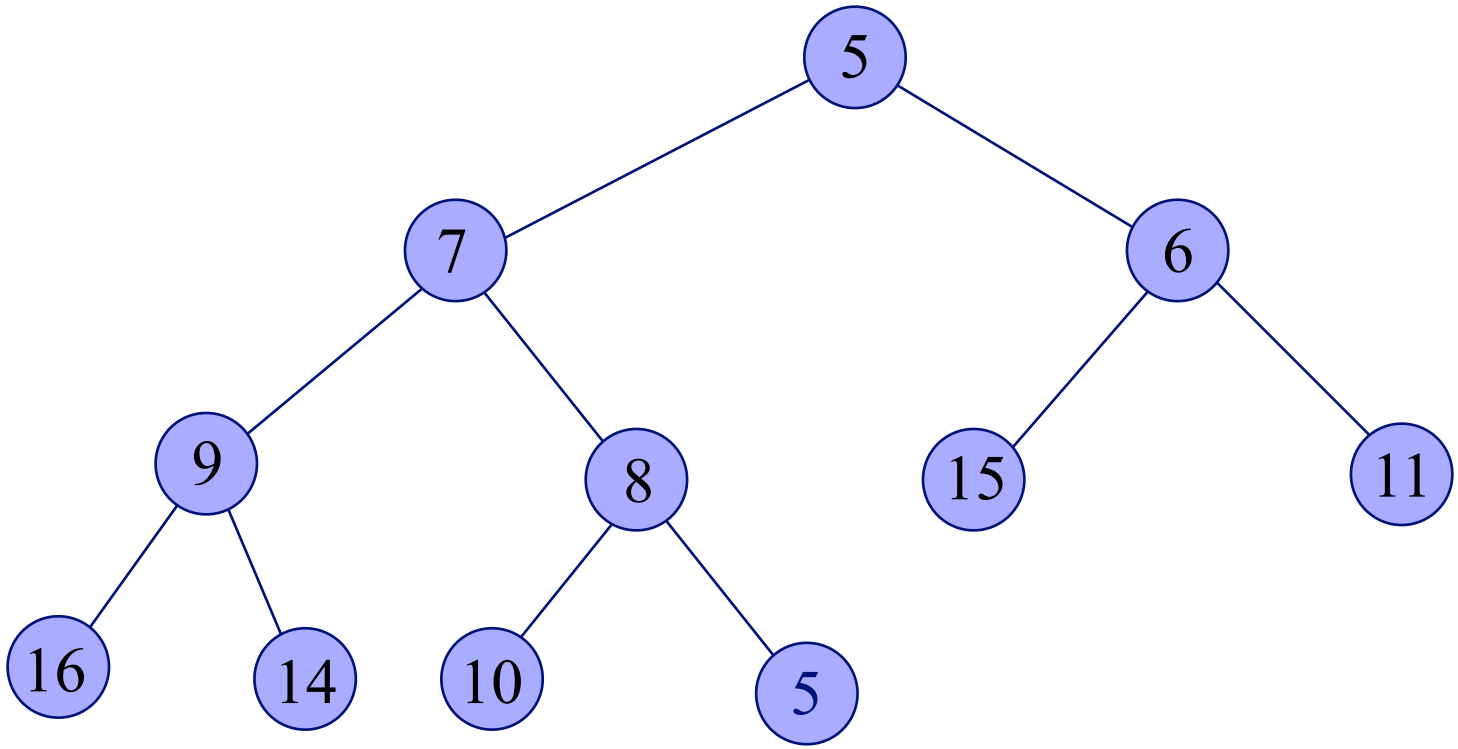
Haldu lze uložit do pole “po hladinách”:



Dítě `a[i]` má rodiče `a[(i-1)//2]`

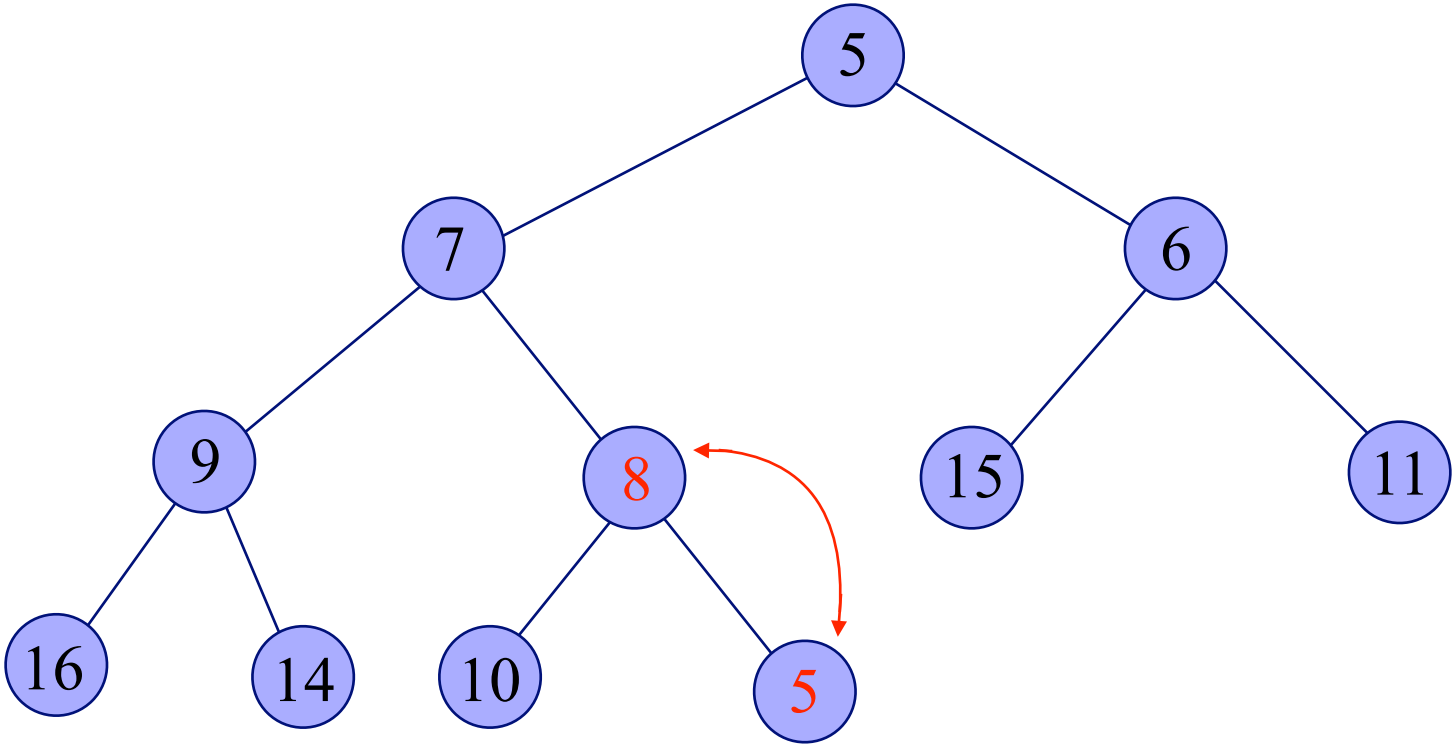
0	1	2	3	4	5	6	7	8	9
5	7	6	9	8	15	11	16	14	10

Binární halda – Přidej



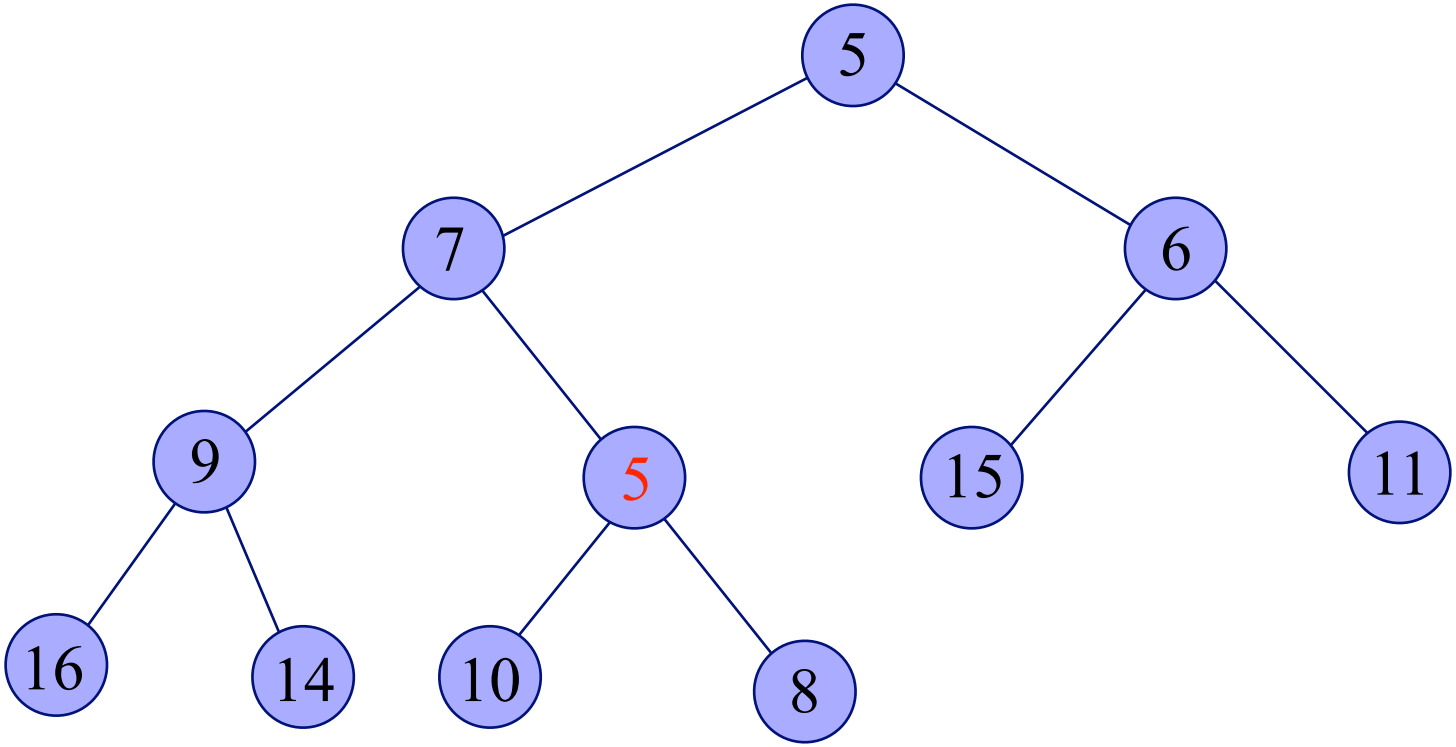
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	5

Binární halda – Přidej



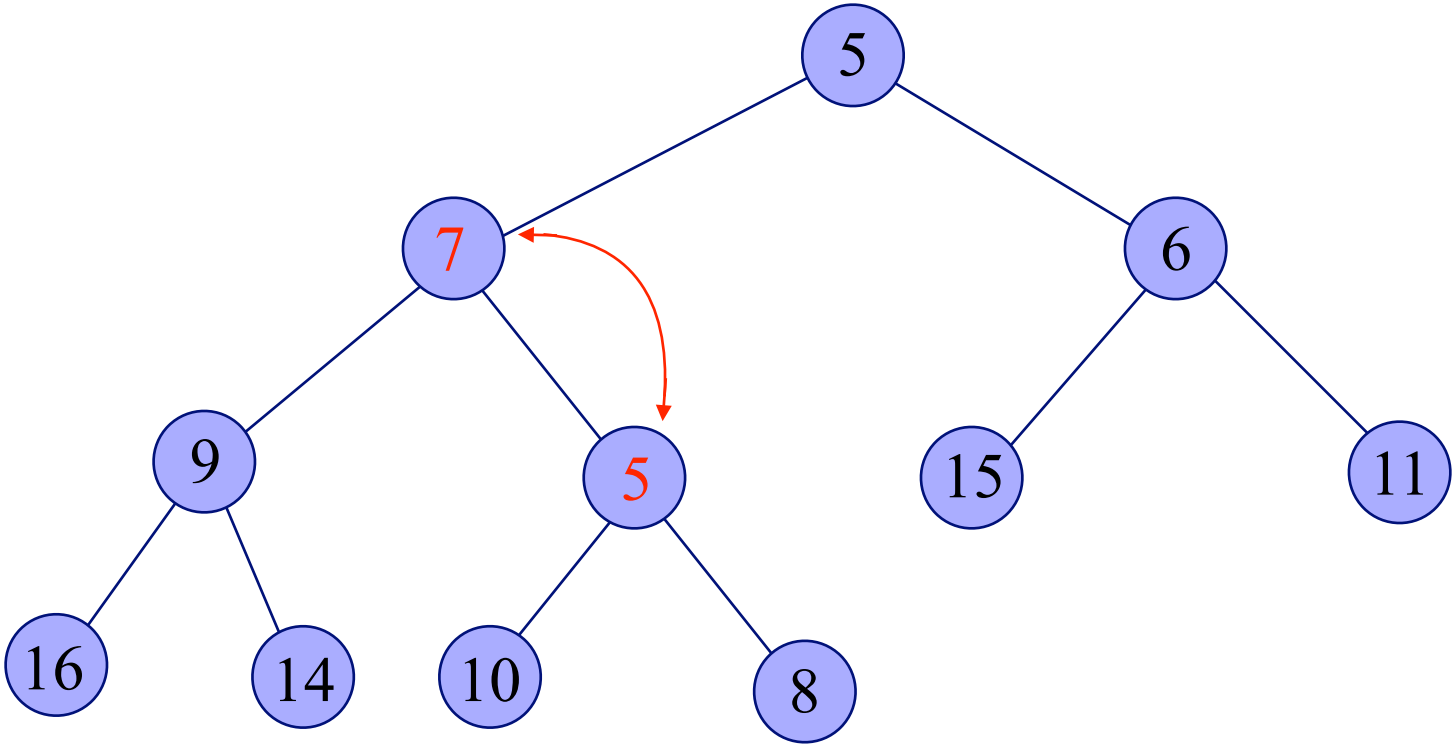
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	5

Binární halda – Přidej



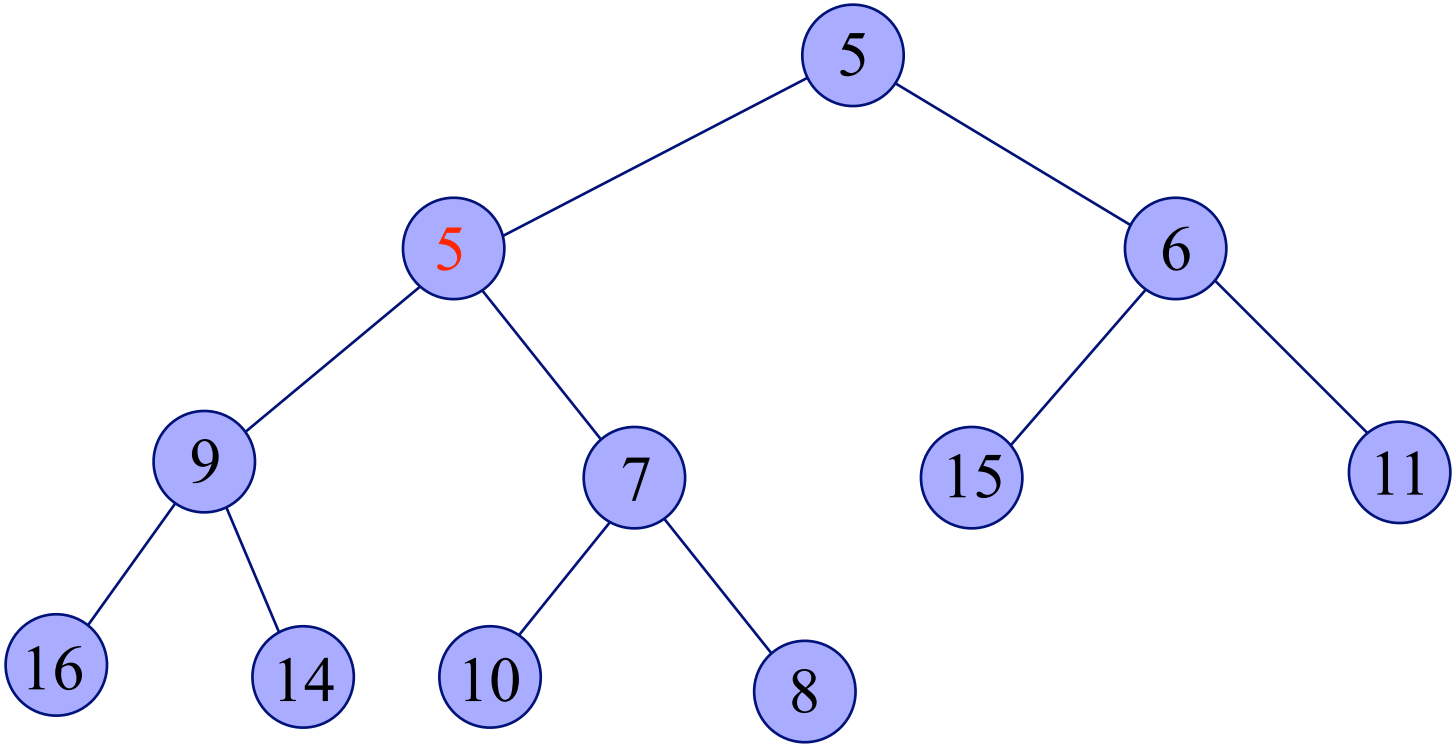
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	5	15	11	16	14	10	8

Binární halda – Přidej



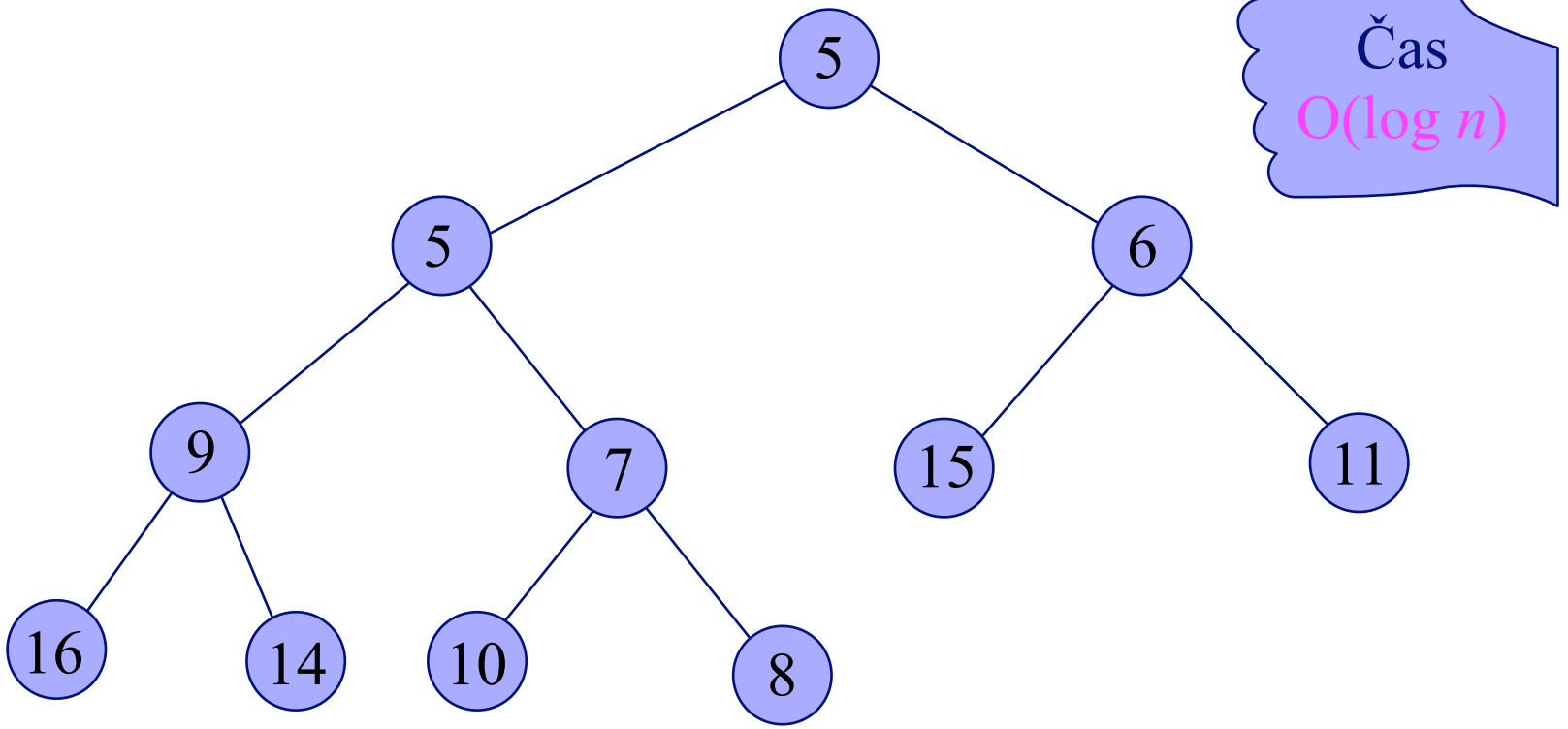
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	5	15	11	16	14	10	8

Binární halda – Přidej



0	1	2	3	4	5	6	7	8	9	10
5	5	6	9	7	15	11	16	14	10	8

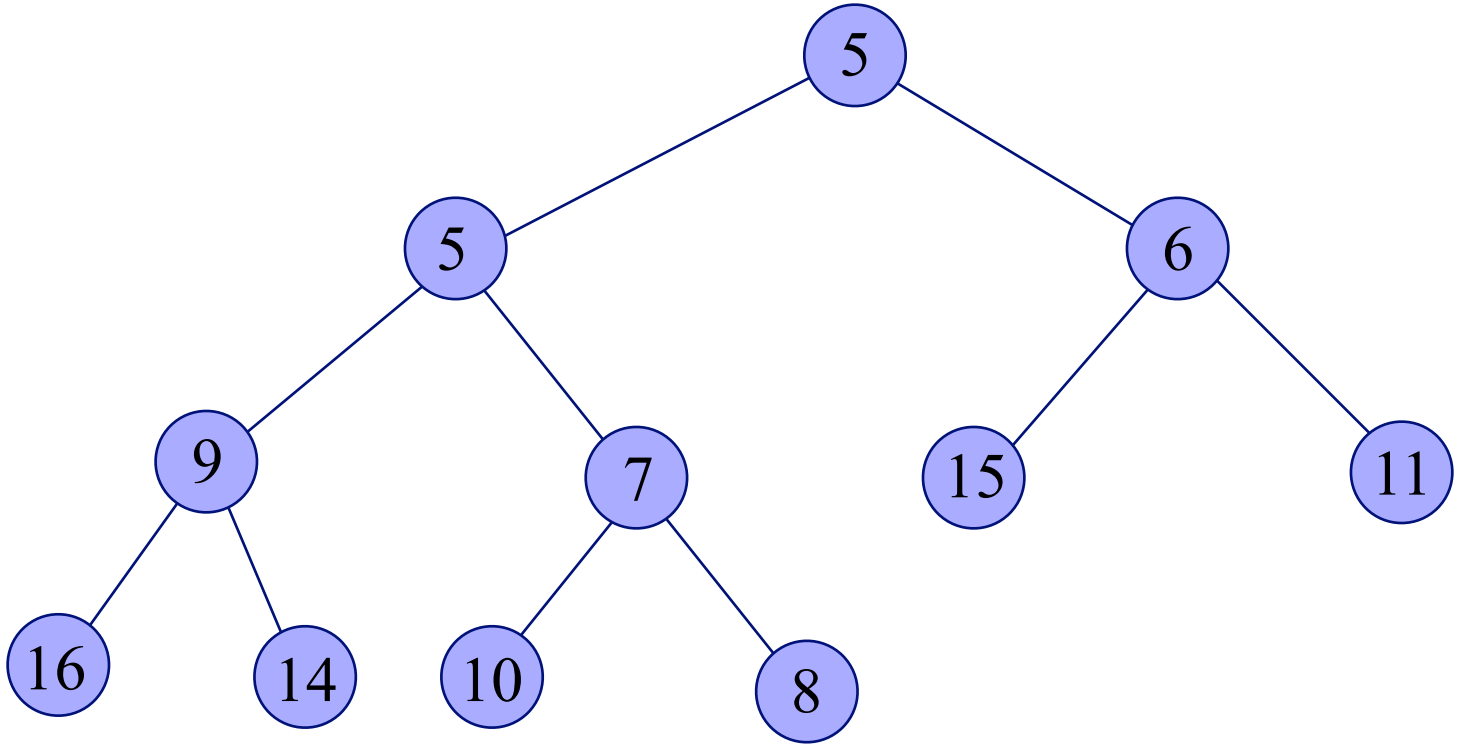
Binární halda – Přidej



Čas
O(log n)

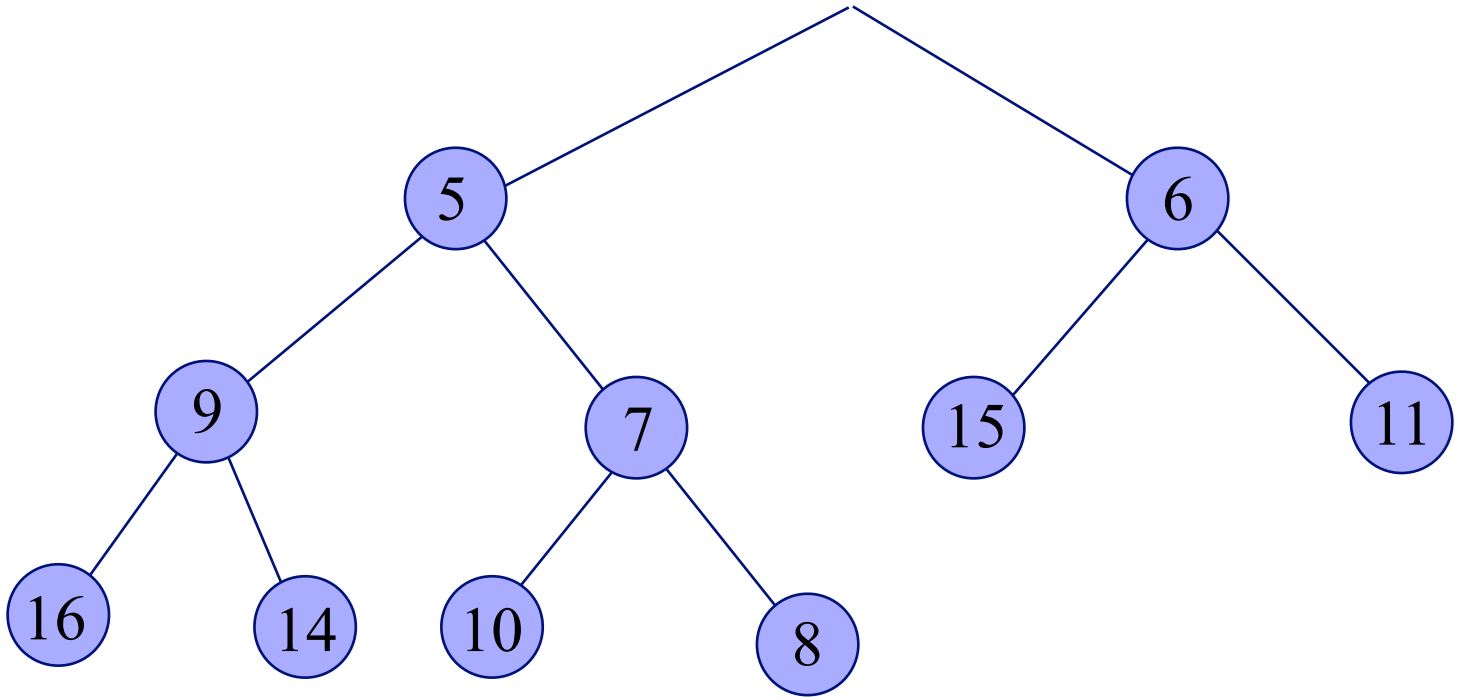
0	1	2	3	4	5	6	7	8	9	10
5	5	6	9	7	15	11	16	14	10	8

Binární halda – OdeberMin



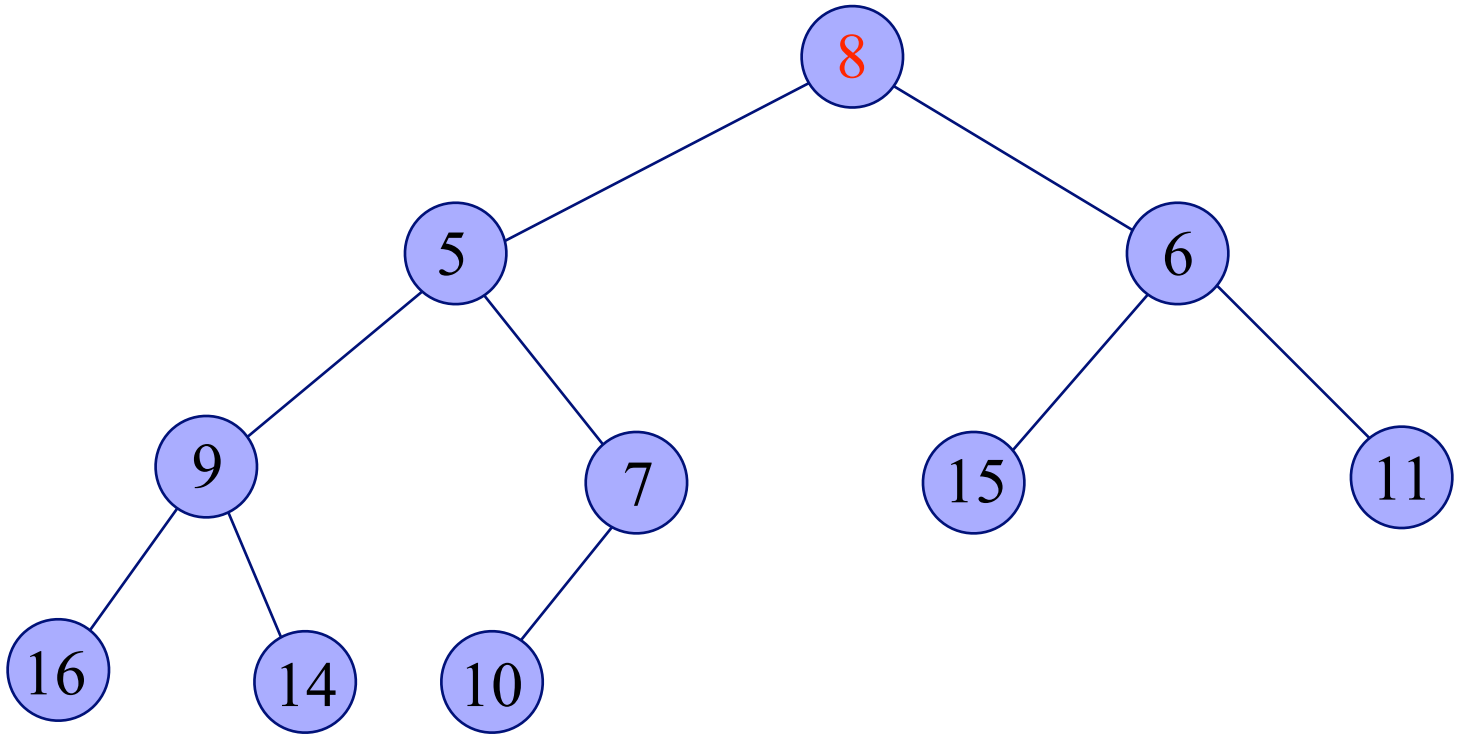
0	1	2	3	4	5	6	7	8	9	10
5	5	6	9	7	15	11	16	14	10	8

Binární halda – OdeberMin



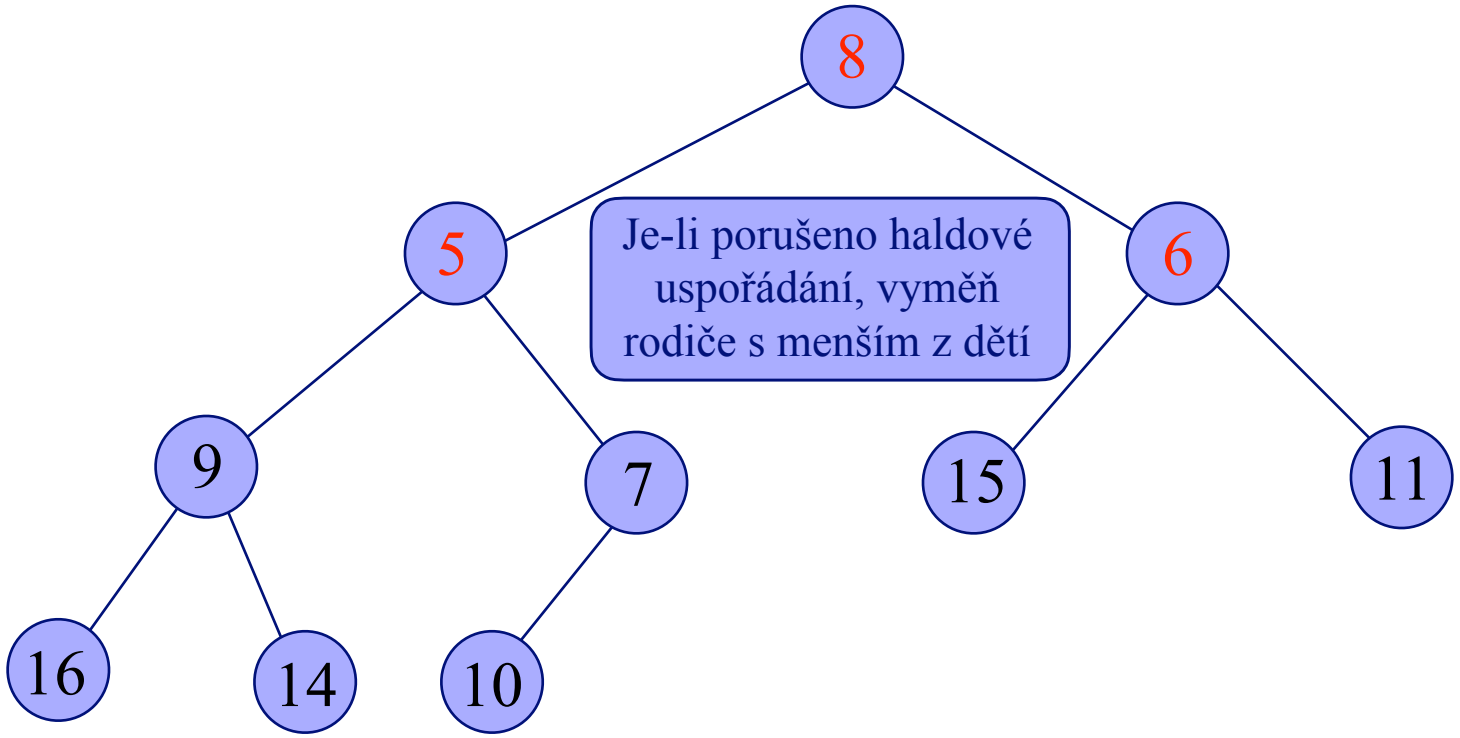
0	1	2	3	4	5	6	7	8	9	10
	5	6	9	7	15	11	16	14	10	8

Binární halda – OdeberMin



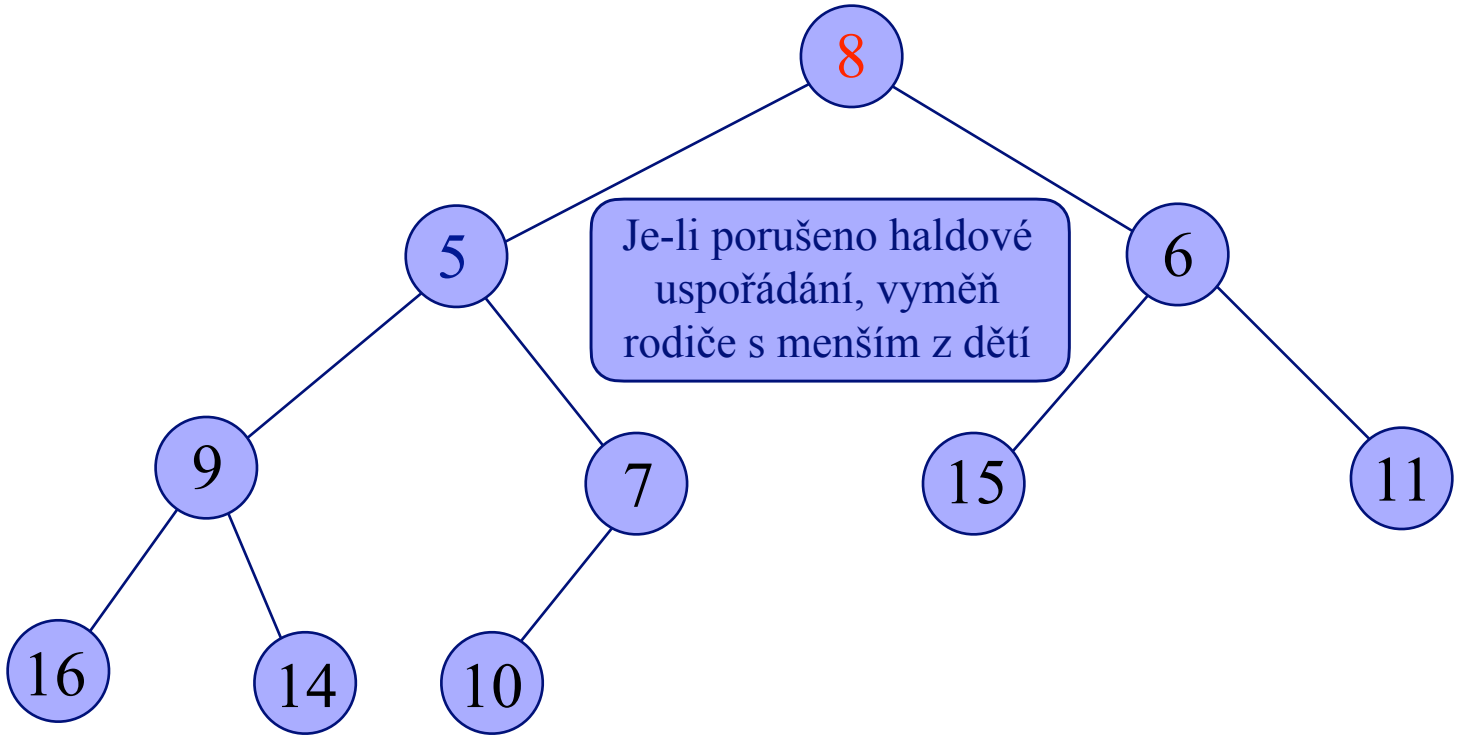
0	1	2	3	4	5	6	7	8	9	10
8	5	6	9	7	15	11	16	14	10	

Binární halda – OdeberMin



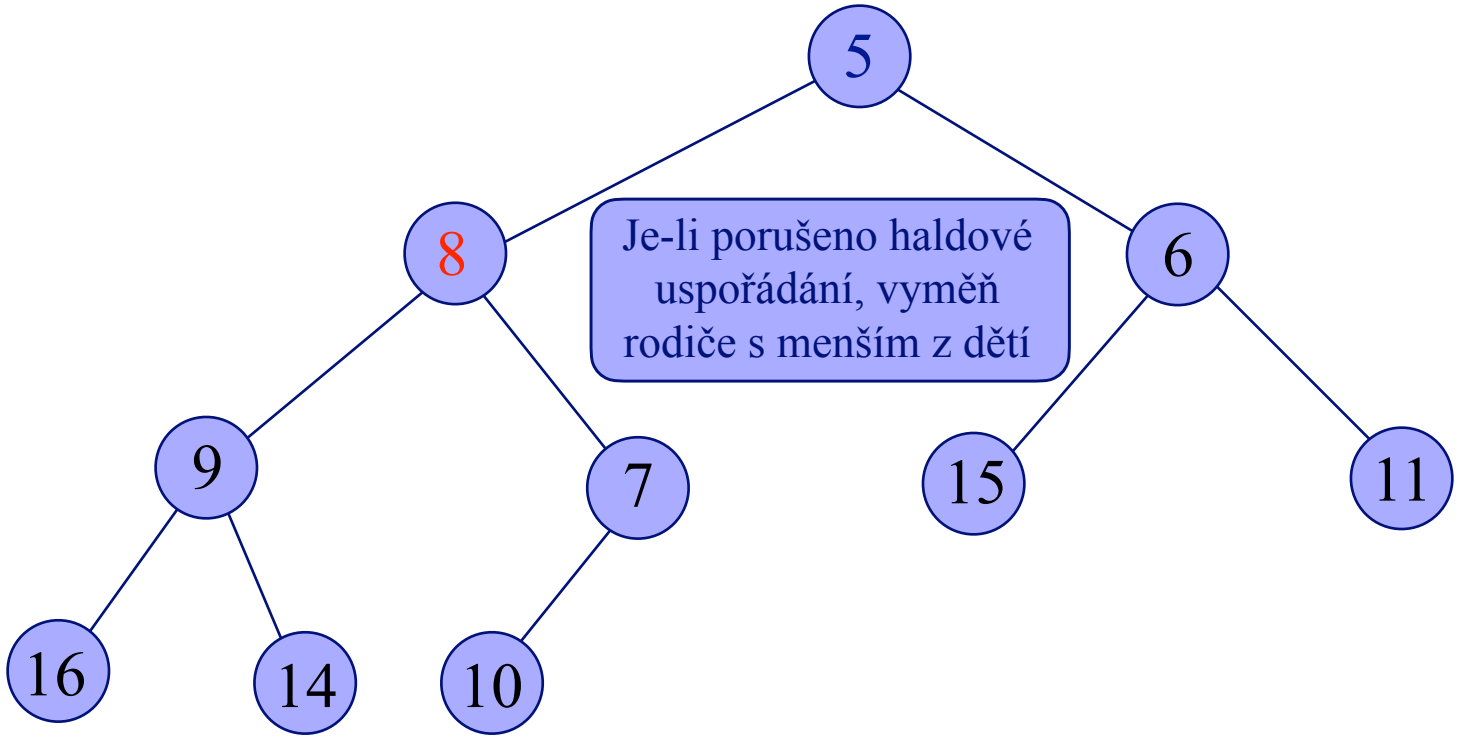
0	1	2	3	4	5	6	7	8	9	10
8	5	6	9	7	15	11	16	14	10	

Binární halda – OdeberMin



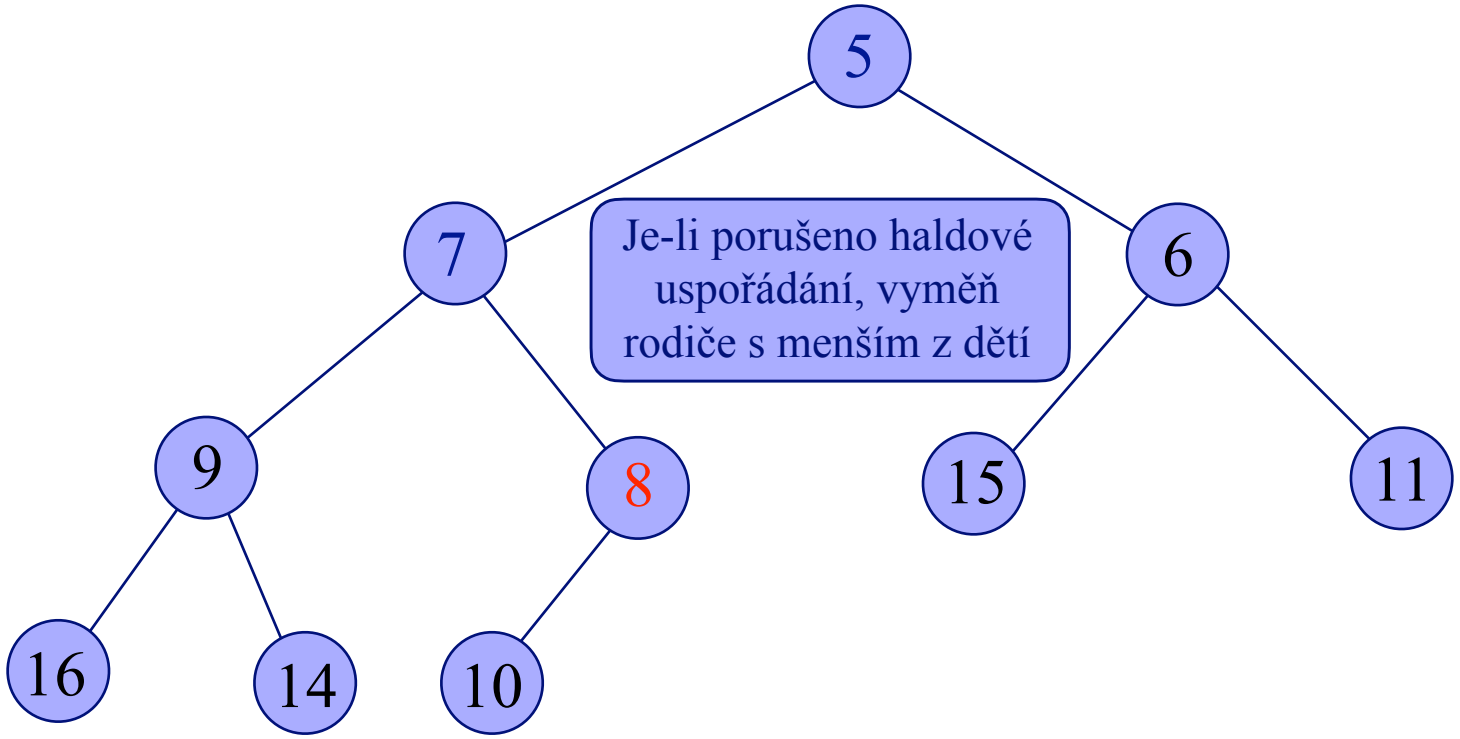
0	1	2	3	4	5	6	7	8	9	10
8	5	6	9	7	15	11	16	14	10	

Binární halda – OdeberMin



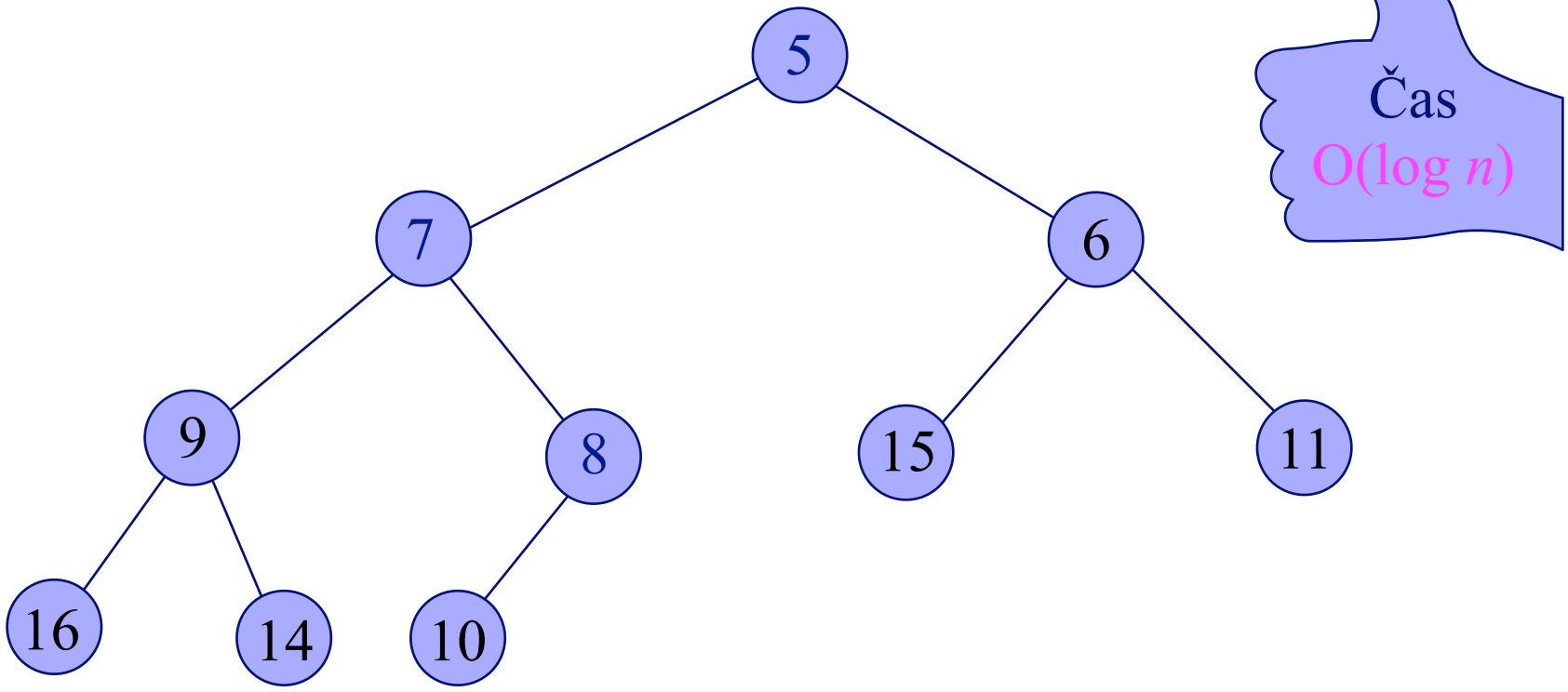
0	1	2	3	4	5	6	7	8	9	10
5	8	6	9	7	15	11	16	14	10	

Binární halda – OdeberMin



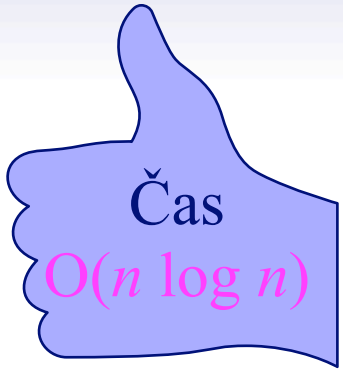
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	

Binární halda – OdeberMin



0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	

Haldové třídění



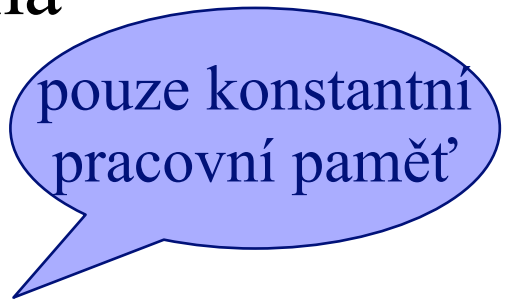
Vstup: pole `a`

① Z prvků pole `a` vybuduj haldu

- začni s triviální haldou obsahující jen `a[0]`
- postupně vkládej `a[1]`, `a[2]`,… pomocí Přidej
- v poli `a` je nyní uložena halda

② Z haldy postupně odebírej minima

- pomocí OdeberMin



✎ **Problém**

- jak zajistíme třídění na místě (in situ) ?

Problémy ✎

② V jazyce Python sestavte funkci

```
heapSort(a)
```

která setřídí prvky zadaného pole `a` vzestupně haldovým tříděním. Váš algoritmus byl měl třídit na místě, tj. může využívat jen konstantní pracovní paměť.