STU FEI

Programovacie techniky

4. O-notácia, stabilné triedenie, heap sort

Porovnanie

PC: 10^8 porovnaní za sekundu
Superpočítač: 10^{12} porovnaní za sekundu
Pre n = 10^6 a 10^9

	Insertion sort (n^2)	Merge sort (n log n)	Quick sort (n log n)
PC	2.8 hod ($n=10^6$) 317 rokov ($n=10^9$)	1 sekunda 18 min	0.6 sek 12 min
Superpočítač	1 sekunda 1 týždeň	okamžite okamžite	okamžite okamžite

Dobrý algoritmus a jeho implementácia sú dôležitejšie ako vlastníctvo superpočítača!

Zdroj: Coursera, Algorithms, Part I

Časová zložitosť

Nech algoritmus vykoná na množine s n prvkami nasledovné počty operácií

	trvanie operácie		počet
op1	0.2	s	12n
op2	0.001	s	n^2
op3	2	s	log n
op4	1000	s	20

Celkový čas výpočtu: $0.2*12n + 0.001*n^2 + 2*\log n + 1000*20$ sekúnd

Čo je určujúci člen časovej zložitosti?

- Pre malé n
- Pre veľké n

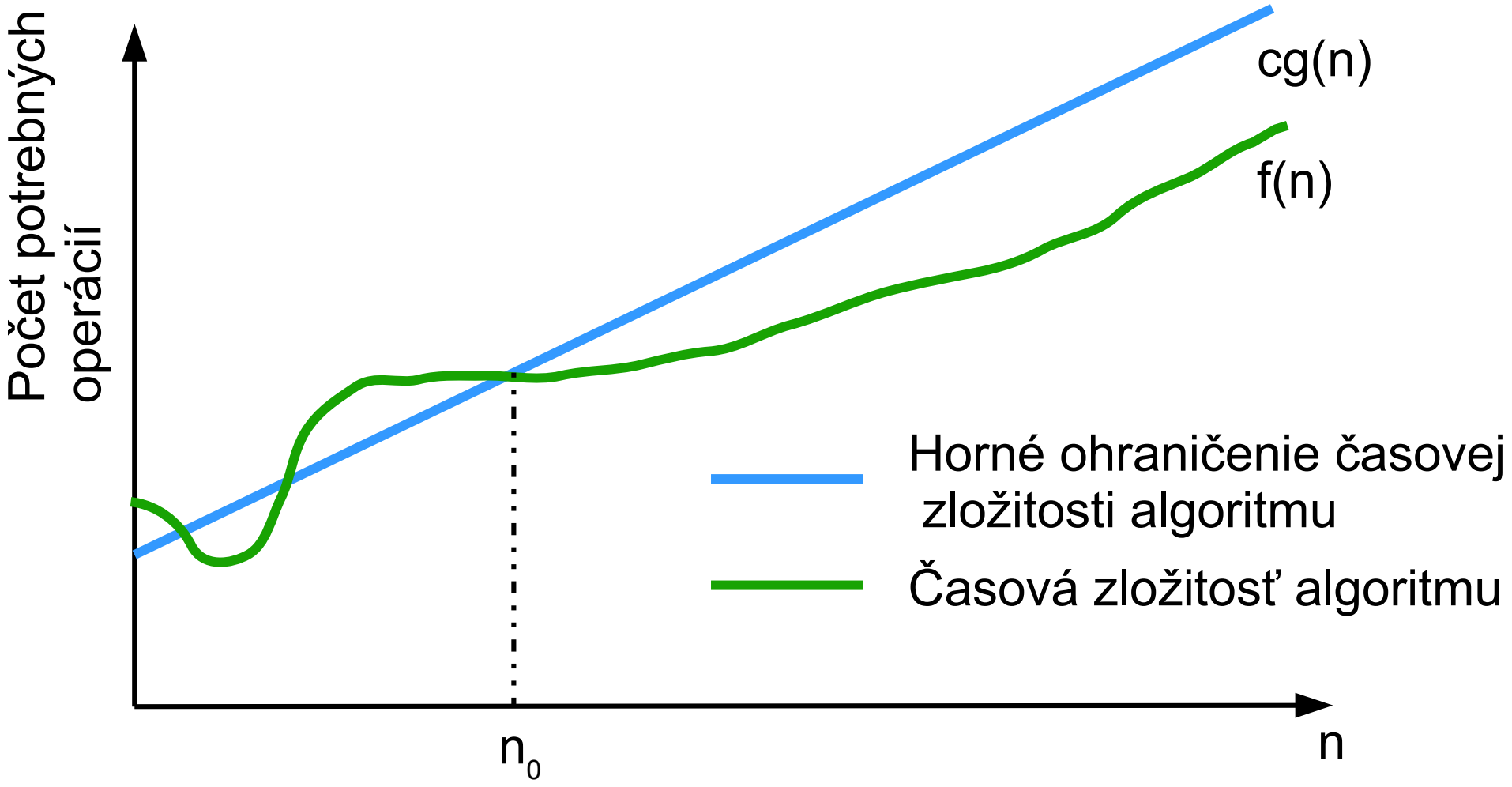
Nájdenie prvku v poli

V najhoršom prípade: n

V priemernom prípade: 0.5n

V najlepšom prípade: 1

The big O notation



O(g(n)): pre $n \geq n_0$ platí, že $0 \leq f(n) \leq cg(n)$,
kde c, n_0 sú kladné konštanty

The big O notation

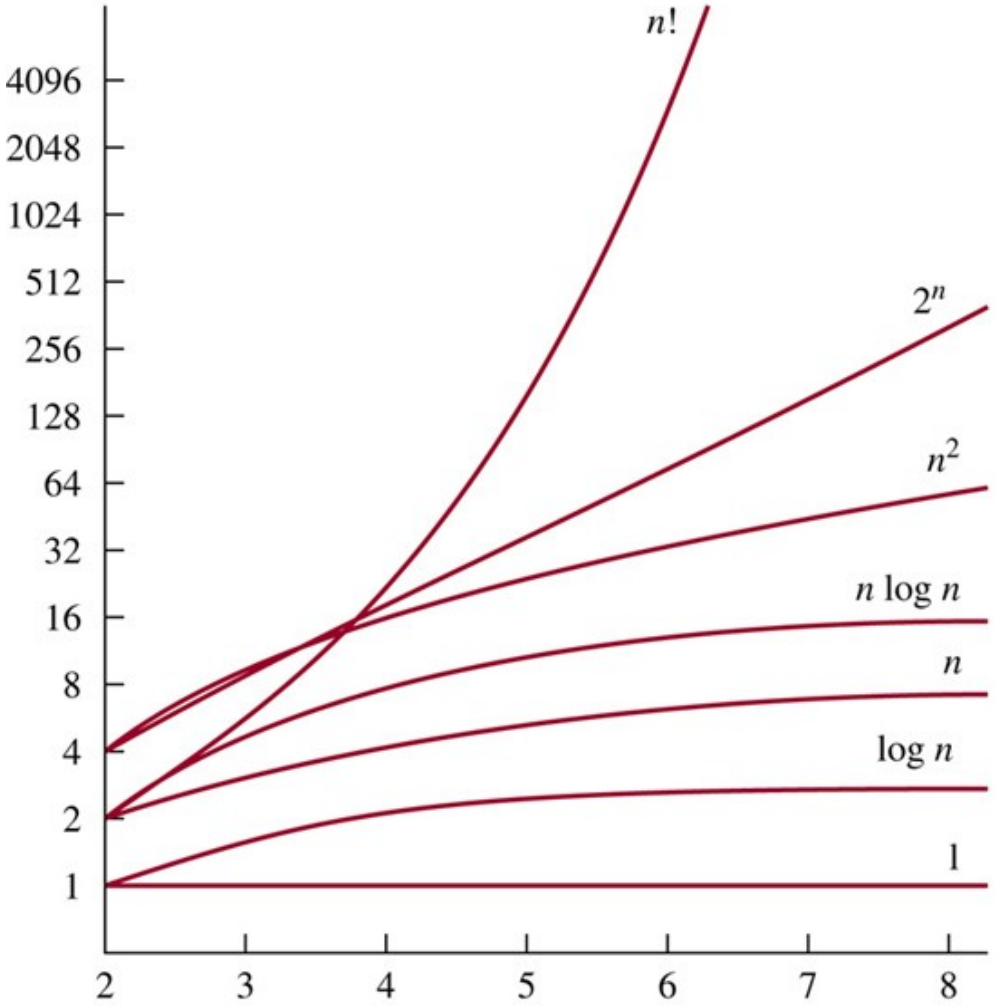
$n^2 = O(n^2)$
$n^2 + n = O(n^2)$
$n^2 + 1000n = O(n^2)$
$5000n^2 + 1000n = O(n^2)$

$n / 1200 = O(n)$
$n^{1.99} = O(n^2)$
$n^2 / \log n = O(n^2)$

$n^2 = O(n^5)$: OK, ale $O(n^2)$ je presnejšie ohraničenie

$n! = O(?)$: ktorá funkcia ohraničuje n! ?

Zložitosť



Zložitosť

O(1): konštantná
O(log n): logaritmická
O(n): lineárna
O(n^2): kvadratická, polynomiálna
O(n^3): kubická, polynomiálna
O(2^n): exponenciálna

O(n!): O(n^n)

Stirling (1730): $$n! \underset{+\infty}{\sim} \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Časová zložitosť: n položiek

	insert	erase	index []	merge	find
Zreťazený zoznam	O(n) vlož na ľub. pozíciu	O(n) zmaž ľub. položku	O(n)	O(1)	O(n)
Zásobník	O(1) push	O(1) pop	O(n)	O(1)	O(n)
Fronta	O(1) enqueue	O(1) dequeue	O(n)	O(1)	O(n)
Pole	O(n) vlož na ľub. pozíciu	O(n) vlož na ľub. pozíciu	O(1)	O(n)	O(n)
Usporiadané pole	O(log n)	O(log n)	O(1)	O(n) merge sort	O(log n)

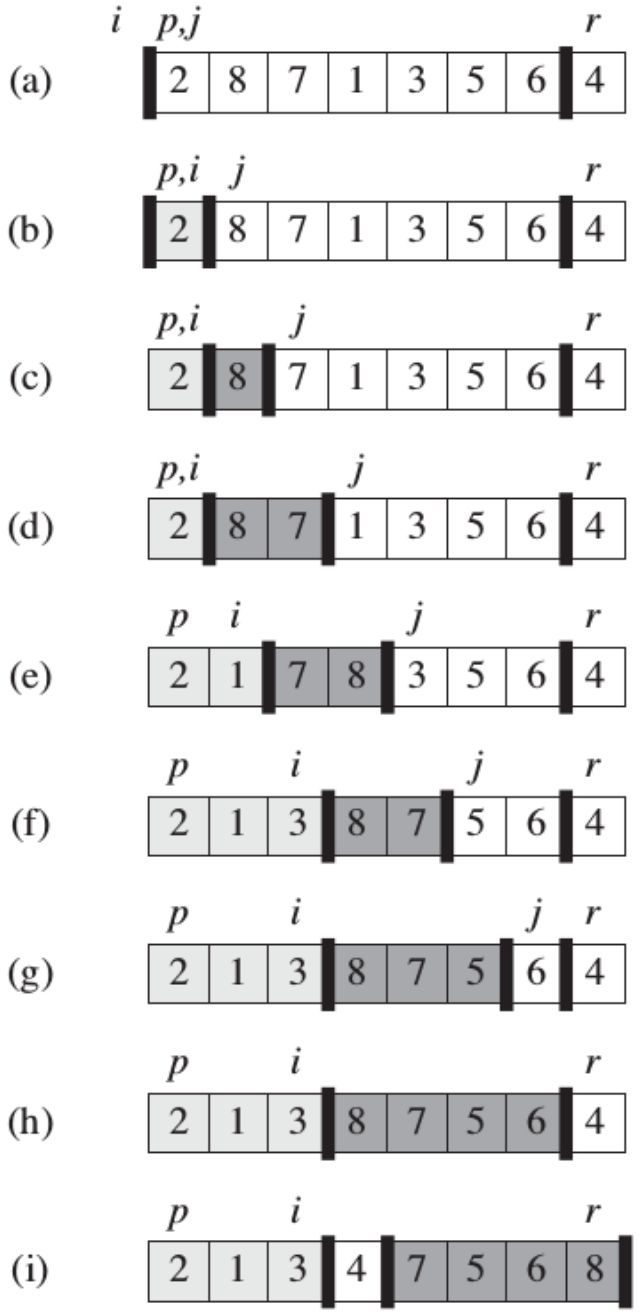
Quick sort: partition

```
PARTITION(A, p, r)
1  x = A[r]                          Pivot je posledný prvok v poli
2  i = p − 1
3  for j = p to r − 1                Prechádzaj cez pole
4      if A[j] ≤ x                   Porovnaj prvok s pivotom
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1                      Vlož pivot, tam kde patrí
```

x je pivot
A je pole A[p..r]

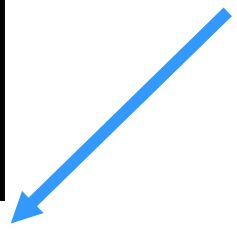
Môže byť quick sort stabilný? Potrebujeme pomocné pole?

Quick sort: partition



Stabilné triedenie

New York	NY
Chicago	IL
Detroit	MI
Buffalo	NY
Milwaukee	WI
Champaign	IL



Chicago	IL
Champaign	IL
Detroit	MI
New York	NY
Buffalo	NY
Milwaukee	WI

Champaign	IL
Chicago	IL
Detroit	MI
New York	NY
Buffalo	NY
Milwaukee	WI

Chicago	IL
Champaign	IL
Detroit	MI
Buffalo	NY
New York	NY
Milwaukee	WI

Champaign	IL
Chicago	IL
Detroit	MI
Buffalo	NY
New York	NY
Milwaukee	WI

Stabilné triedenie = relatívne poradie rovnakých prvkov je zachované.

Stabilné triedenie

A je pole
i, j sú indexové premenné
π(i), π(j): pozícia A[i], A[j] po triedení

Ak i < j a A[i]=A[j],
potom počas a po triedení platí:

π(i) < π(j)

Stabilné triedenie

Bubble/insertion sort: $O(n^2)$

Merge sort: $O(n \log n)$

Prečo? Zachovať relatívne poradie počas merge fázy je jednoduché.

Quicksort nie je stabilný, stabilné verzie existujú, ale nie sú praktické (vyžadujú pomocné pole atď.)

C++: triediaci algoritmus

C++11: O(n log n) v najhoršom prípade

Štandard C++ nepredpisuje špecifický triediaci algoritmus

C++ obsahuje implementácie:

- sort()
- stable_sort() //iná časová zložitosť ako sort()

Heap sort

Časová zložitosť O(n log n)

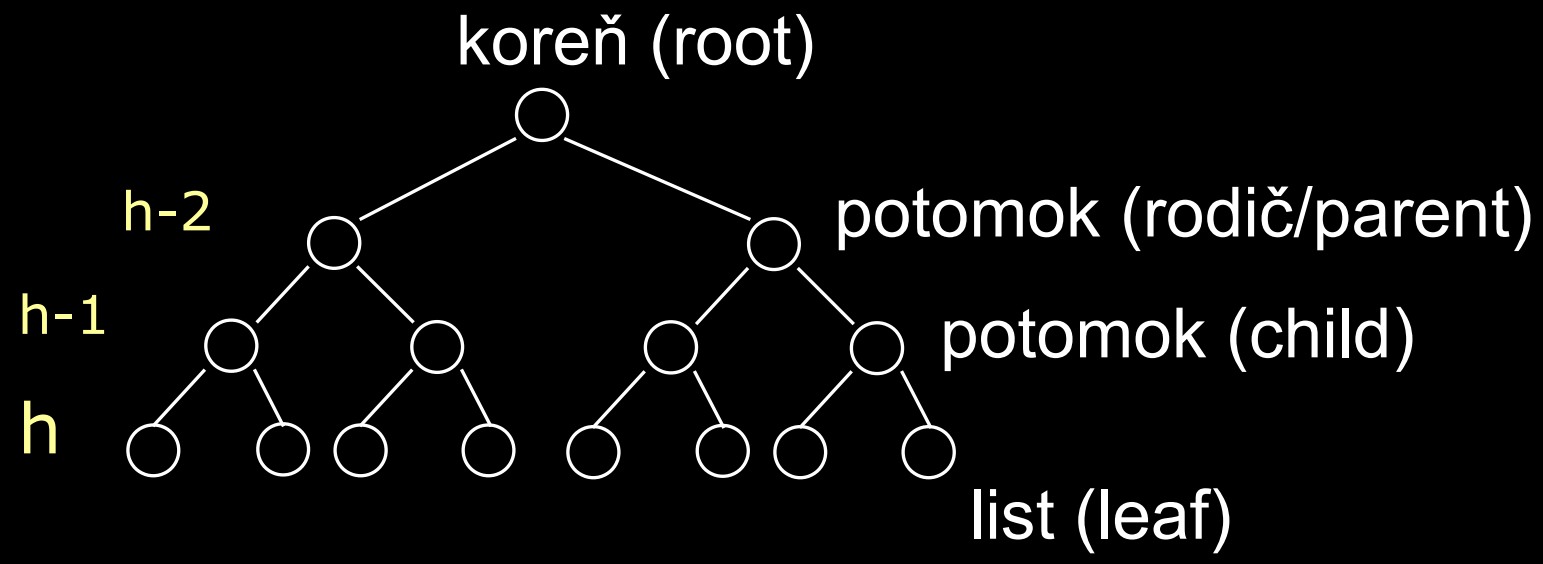
Quick sort je v najhoršom prípade $O(n^2)$, v priemere O(n log n)

Heap sort je lepší kandidát v časovo kritických prípadoch, ale Quick sort je väčšinou rýchlejší

Heap sort: J. W. J. Williams

1964: foto je ťažko nájsť

Binárny strom s hĺbkou h



Uzol: koreň, potomok, list...
Binárny strom: každý uzol, okrem listov, má dvoch potomkov

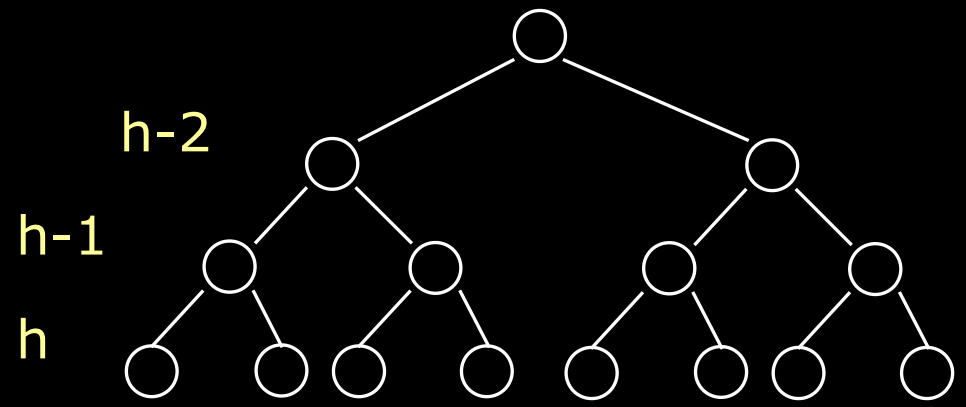
Vyvážený binárny strom

Hĺbka uzla: vzdialenosť od koreňa

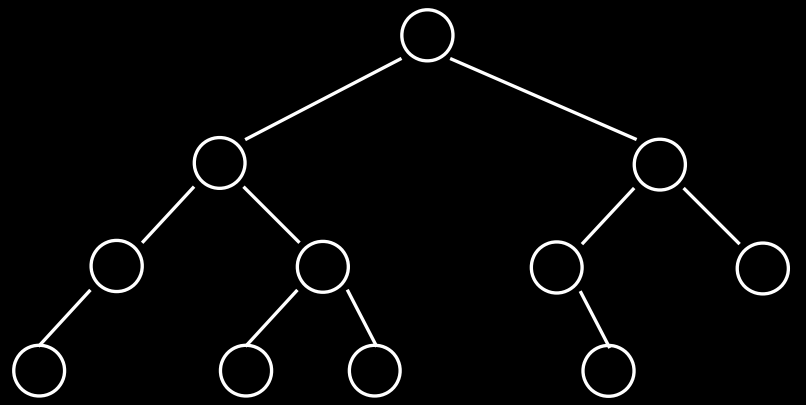
Hĺbka stromu: vzdialenosť najvzdialenejšieho uzla od koreňa

Binárny strom s hĺbkou *h* je vyvážený, ak všetky jeho uzly s hĺbkou do h-2 majú dvoch potomkov

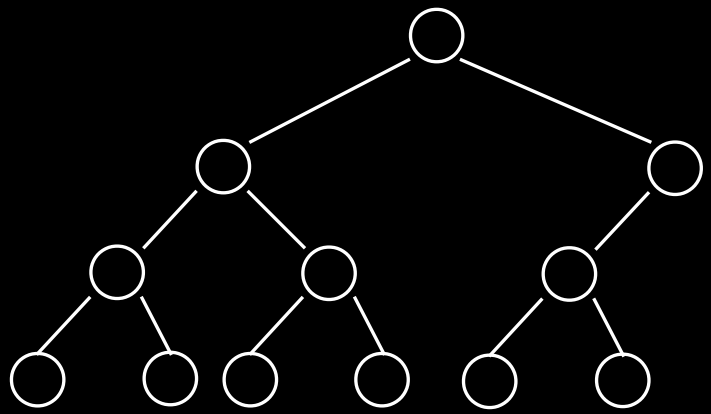
Vyvážený binárny strom



Vyvážený



Vyvážený



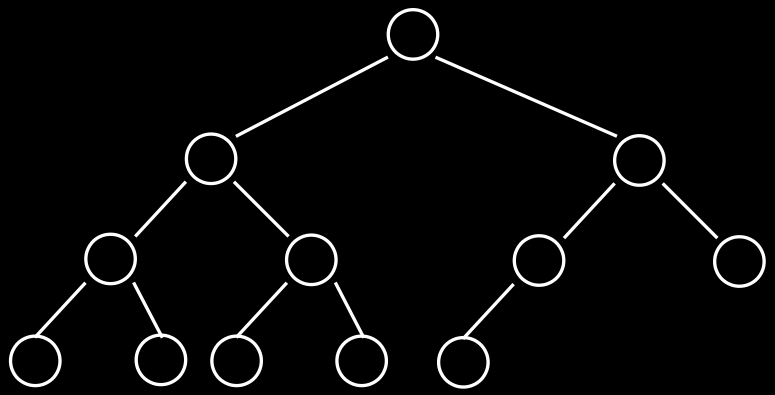
Nevyvážený

Doľava zarovnaný vyvážený..

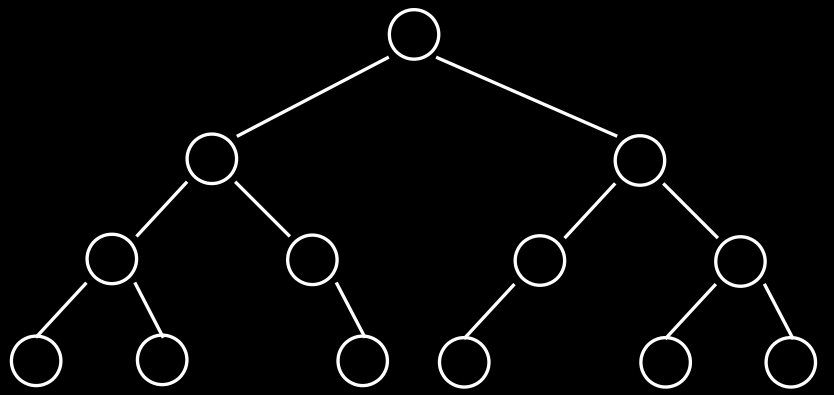
Vyvážený binárny strom je doľava zarovnaný ak:

- všetky listy majú rovnakú hĺbku, alebo
- všetky listy s hĺbkou h sú vľavo od listov s hĺbkou h-1

Vyvážený binárny strom



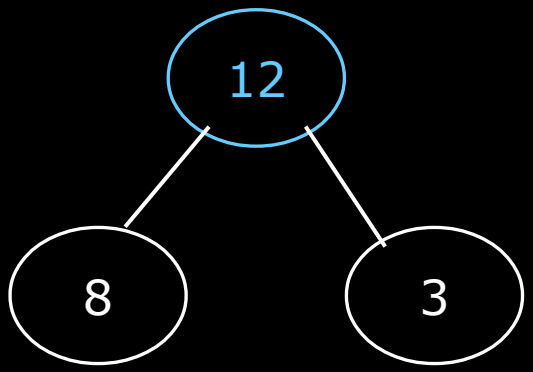
Doľava zarovnaný



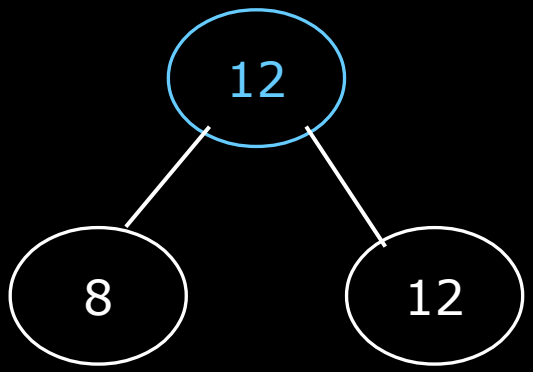
Nie je doľava zarovnaný

Heap vlastnosť

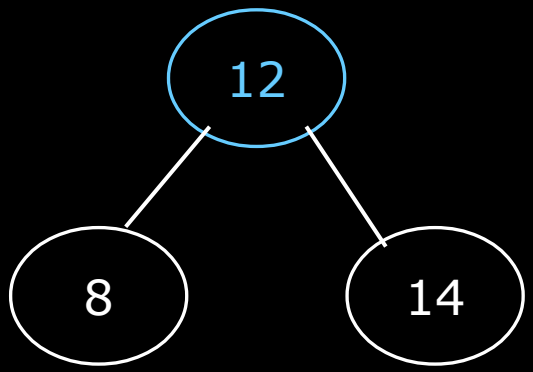
Hodnota v uzle je aspoň taká ako hodnota v potomkoch



Vlastnosť splnená



Vlastnosť splnená



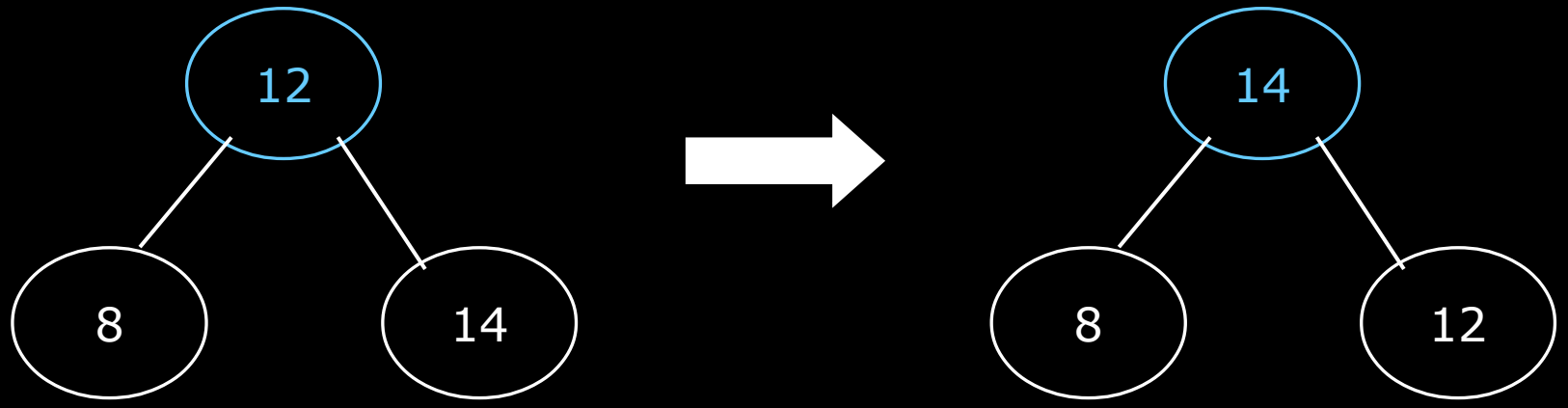
Vlastnosť nie je splnená

Binárny strom je heap, ak je vlastnosť splnená pre všetky uzly

Heap = halda, kopa, hromada...

SiftUp

Porovnaj hodnotu uzla s hodnotou rodiča, vymeň ak má väčšiu hodnotu



Podmienka nie je splnená

Podmienka je splnená

Takáto výmena sa volá „sifting up“

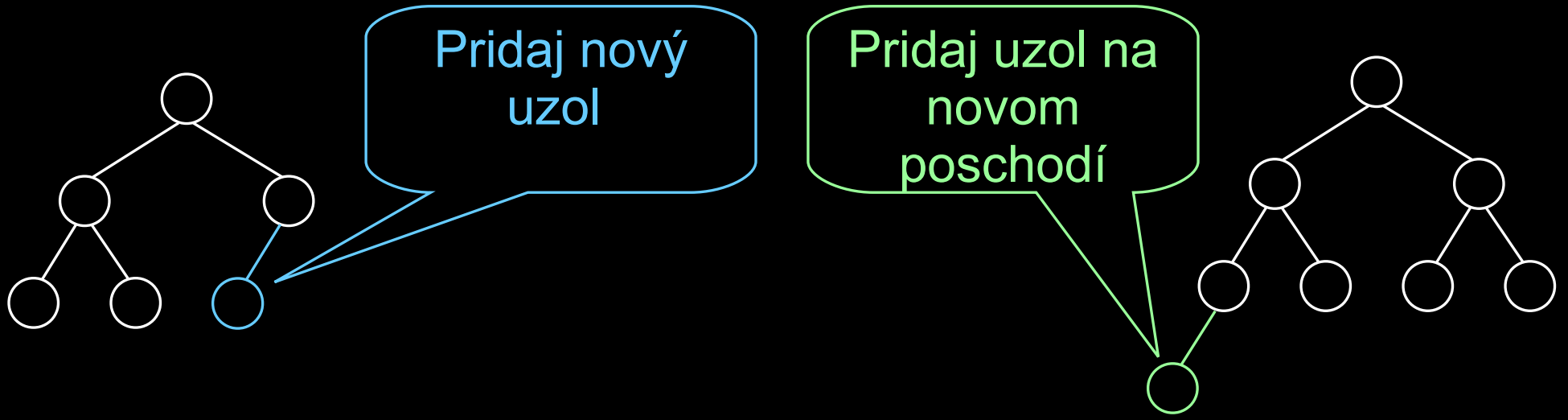
Po výmene nemusí byť heap vlastnosť stromu splnená

Heap: konštrukcia

Strom, ktorý má jeden uzol je heap

Ak treba pridať uzol:

- Pridávaj uzol na najhlbšom poschodí na pravo od uzlu čo je najviac vpravo
- Ak na poschodí nie je voľné miesto, začni na novom hlbšom poschodí



Heap: konštrukcia pokr.

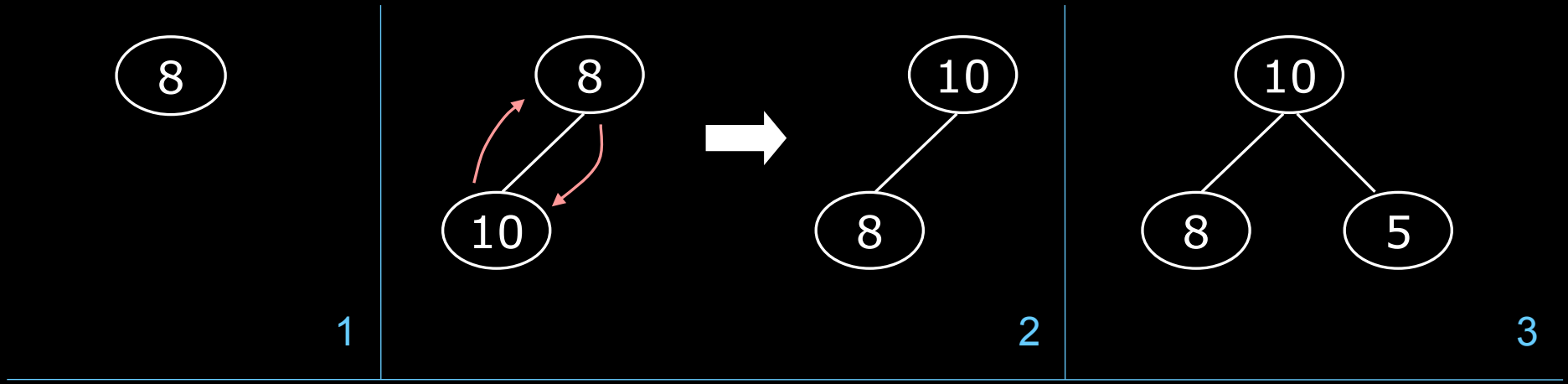
Vždy keď pridáme uzol môžeme porušiť heap podmienku

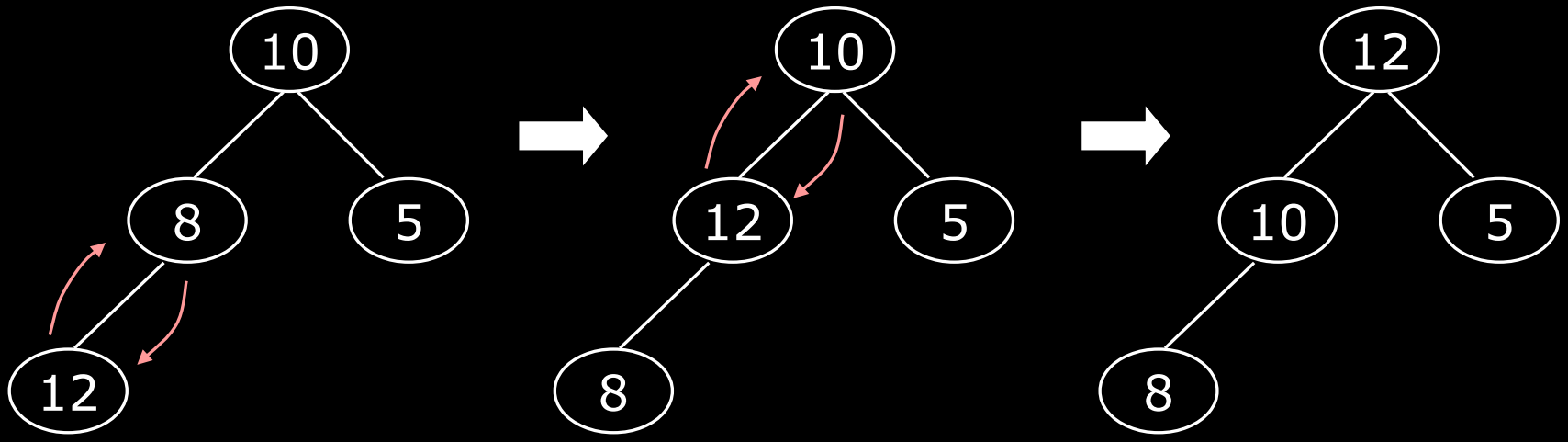
Vždy keď pridáme uzol a porušíme heap vlastnosť musíme urobiť SiftUp

Ak urobíme SiftUp možeme porušiť heap vlastnosť pre rodiča

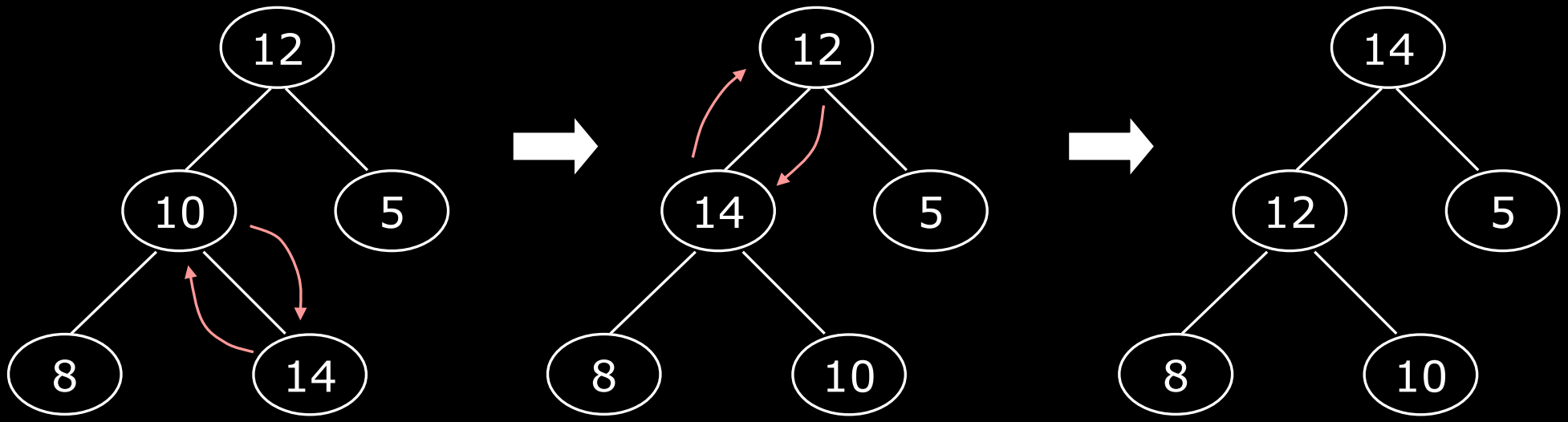
SiftUp opakujeme až pokým už nie je potrebný, alebo ak dosiahneme koreň

Heap: konštrukcia príklad

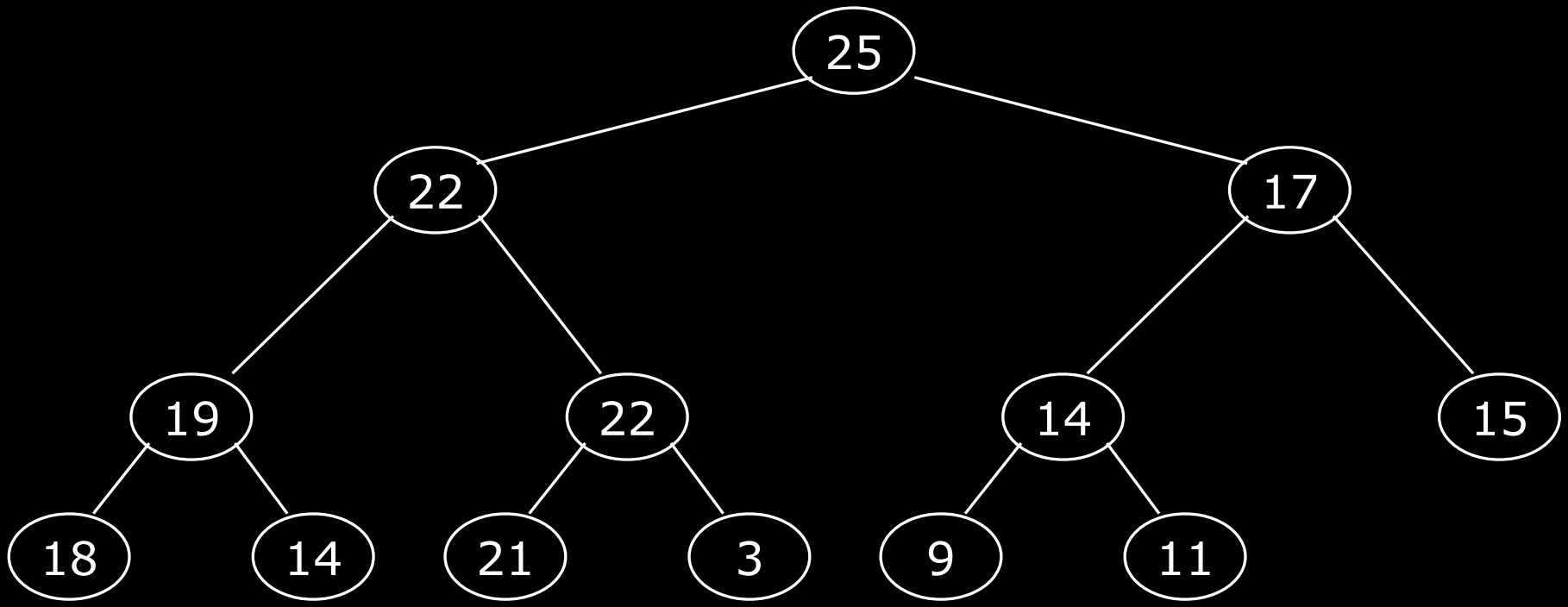




Heap: konštrukcia príklad



Heap: konštrukcia príklad

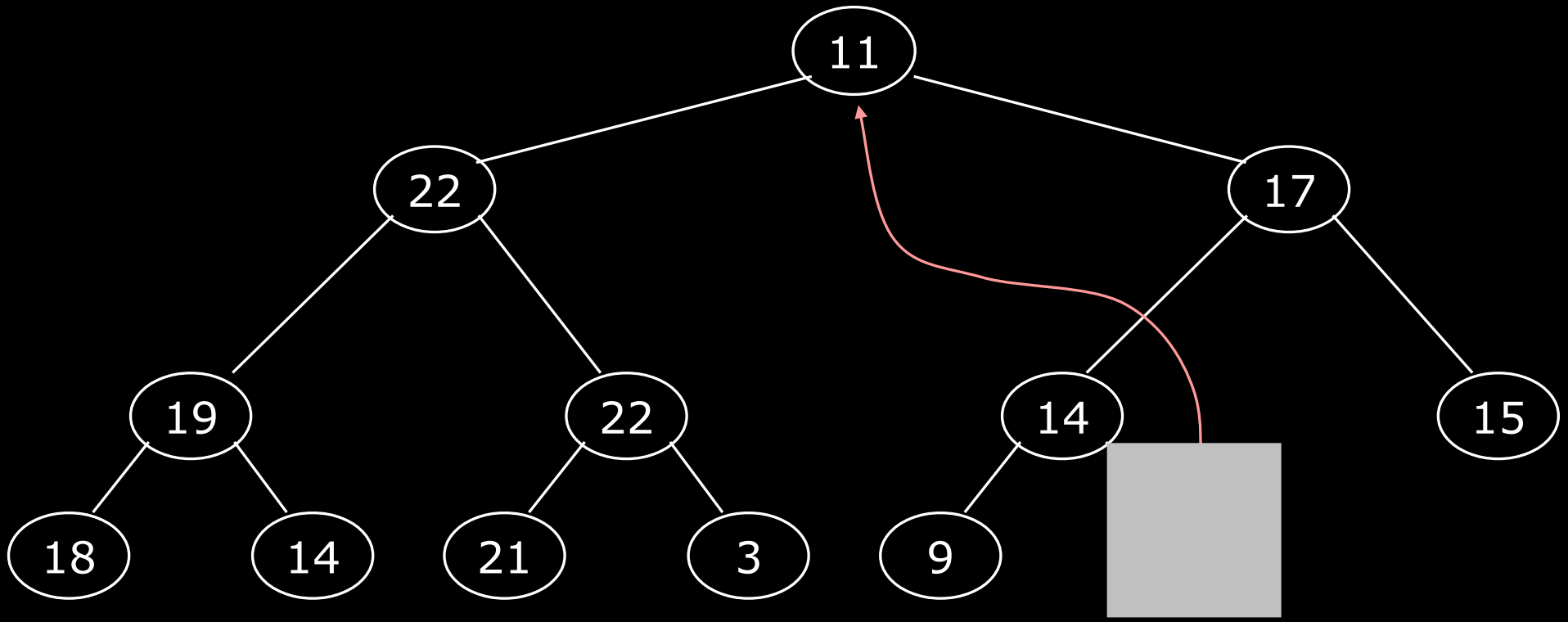


Binárny strom, ktorý spĺňa heap vlastnosť

Heap nie je roztriedený

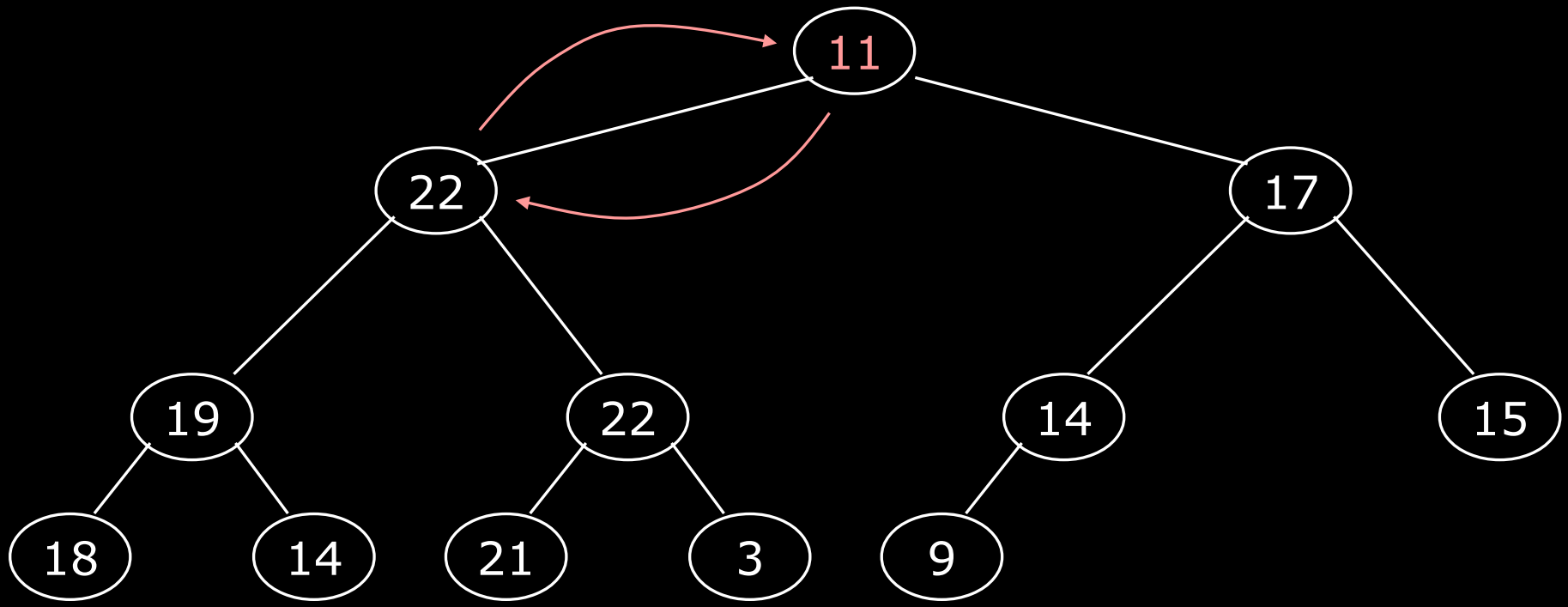
SiftUp nemení vzhľad binárneho stromu

Heap: odstránenie koreňa



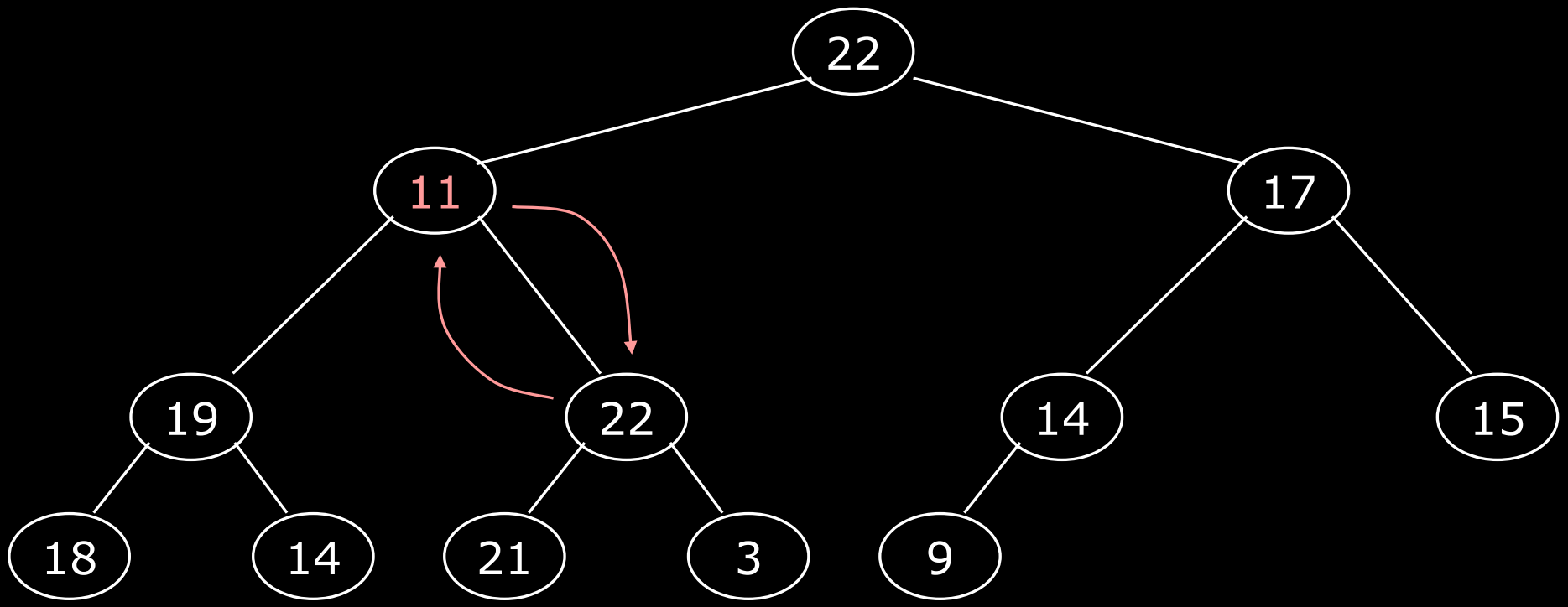
Čo musíme spraviť aby bol binárny strom vyvážený a doľava zarovnaný? : list najviac vpravo je nový koreň

Sift down (re-heap)

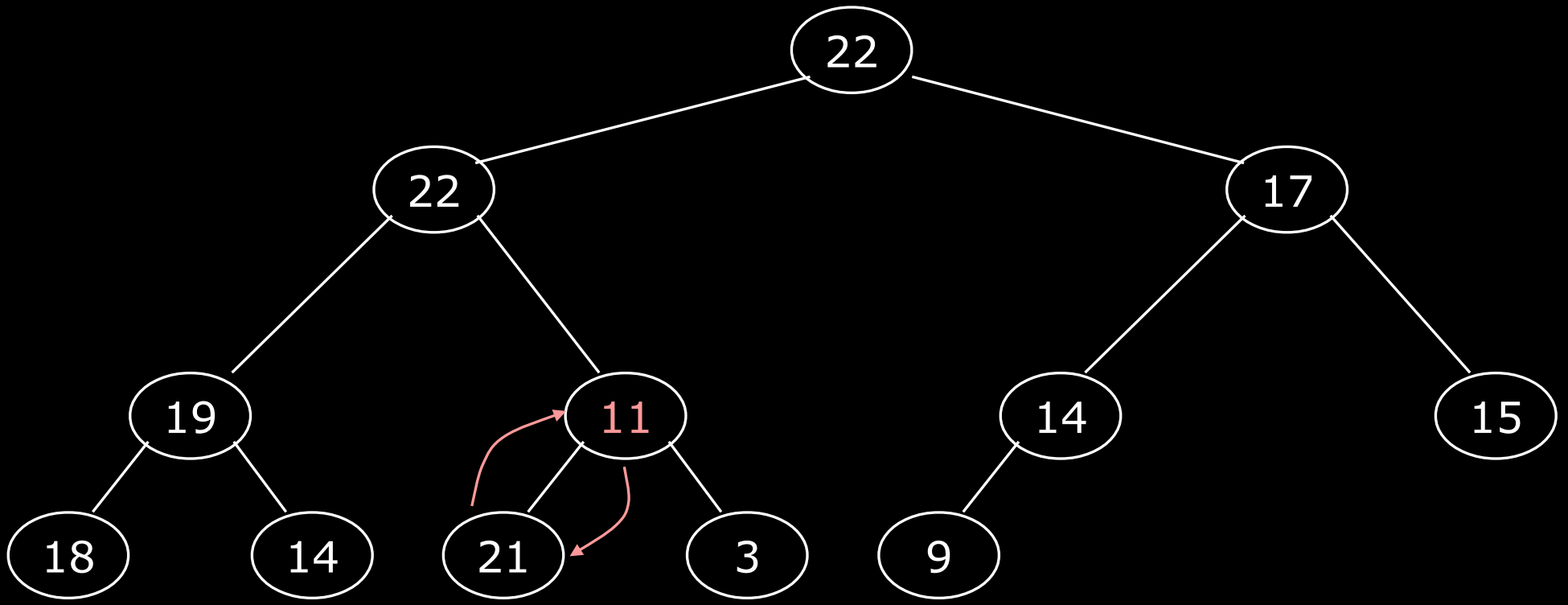


Koreň stratil heap vlastnosť: porovnaj s potomkami. Ak je heap podmienka porušená, potom vymeň s potomkom, ktorý je väčší.

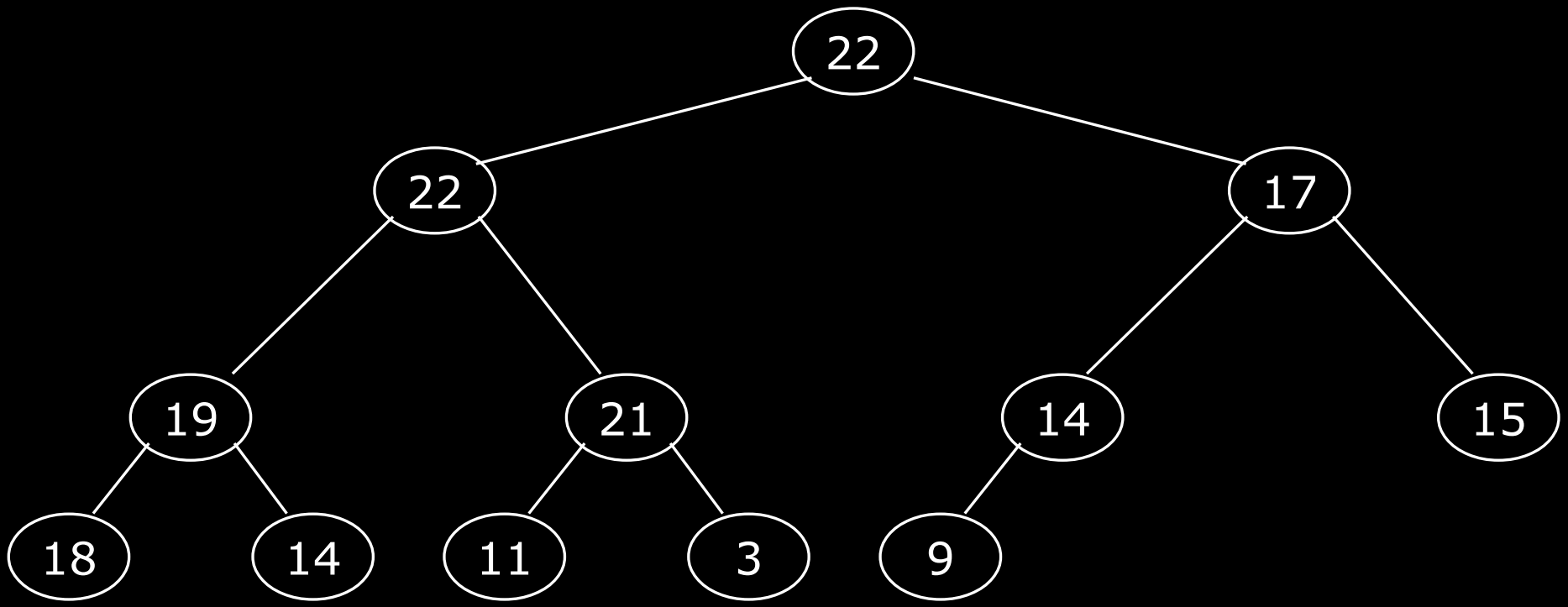
Sift down



Sift down



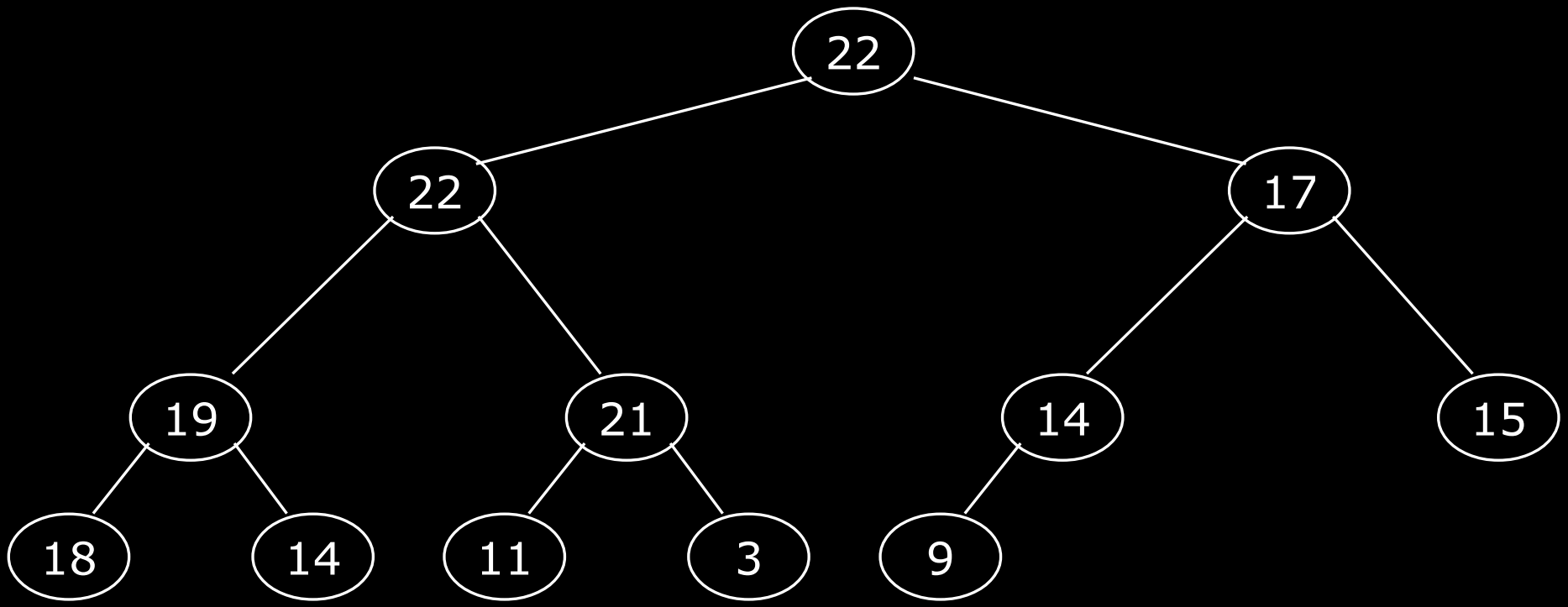
Heap



Koreň má najväčšiu hodnotu!!

22...(aha...)

Heap

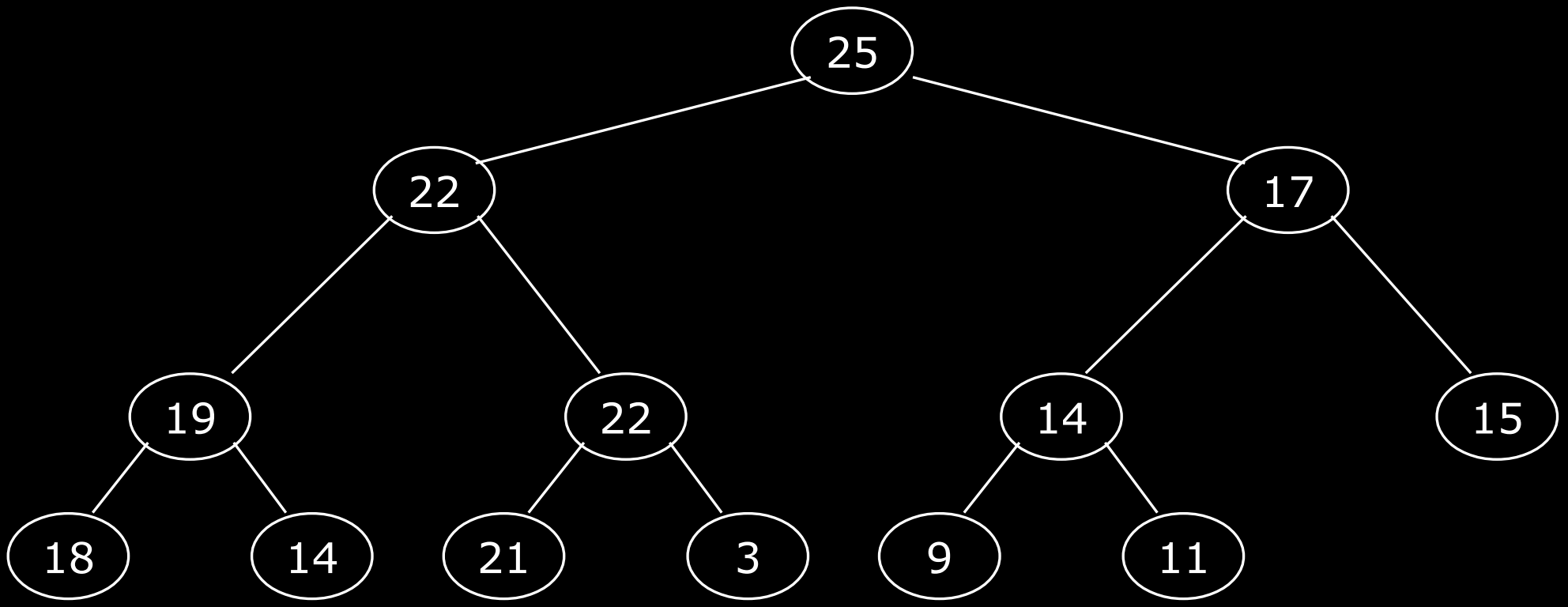


Odstraňuj koreň až pokým binárny strom nie je prázdny!

Heap: triedenie

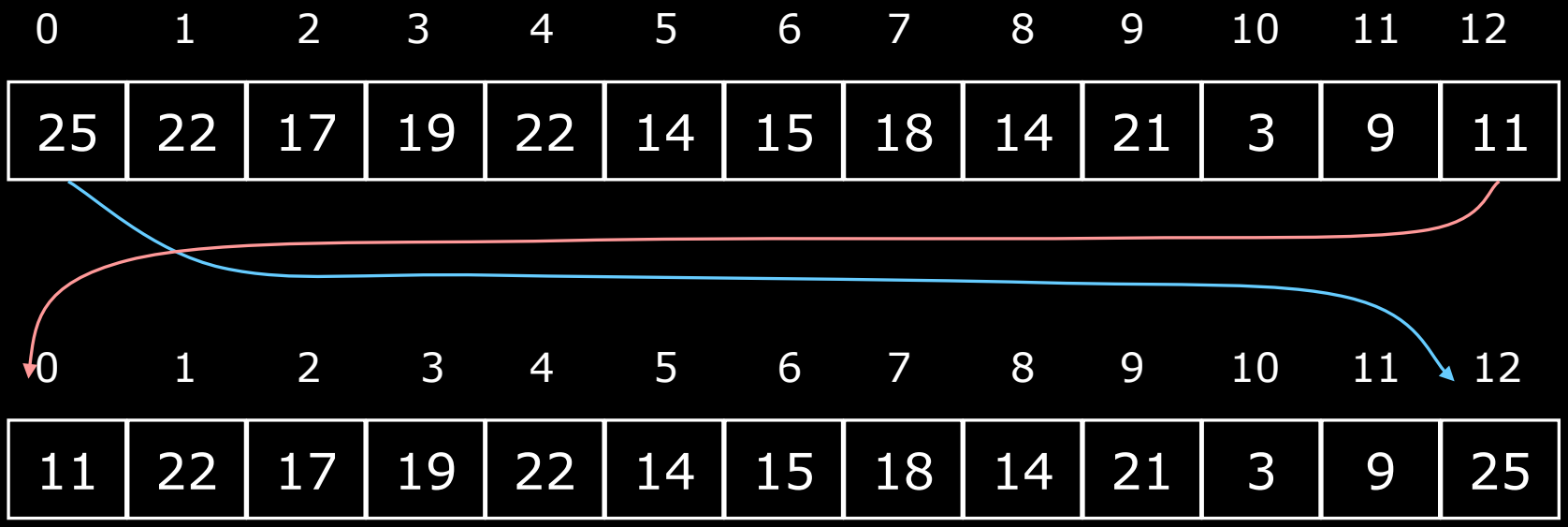
```
urob z poľa heap;

opakuj {
    odstráň a nahraď koreň;
    reHeap;
}
```

Pole ako heap



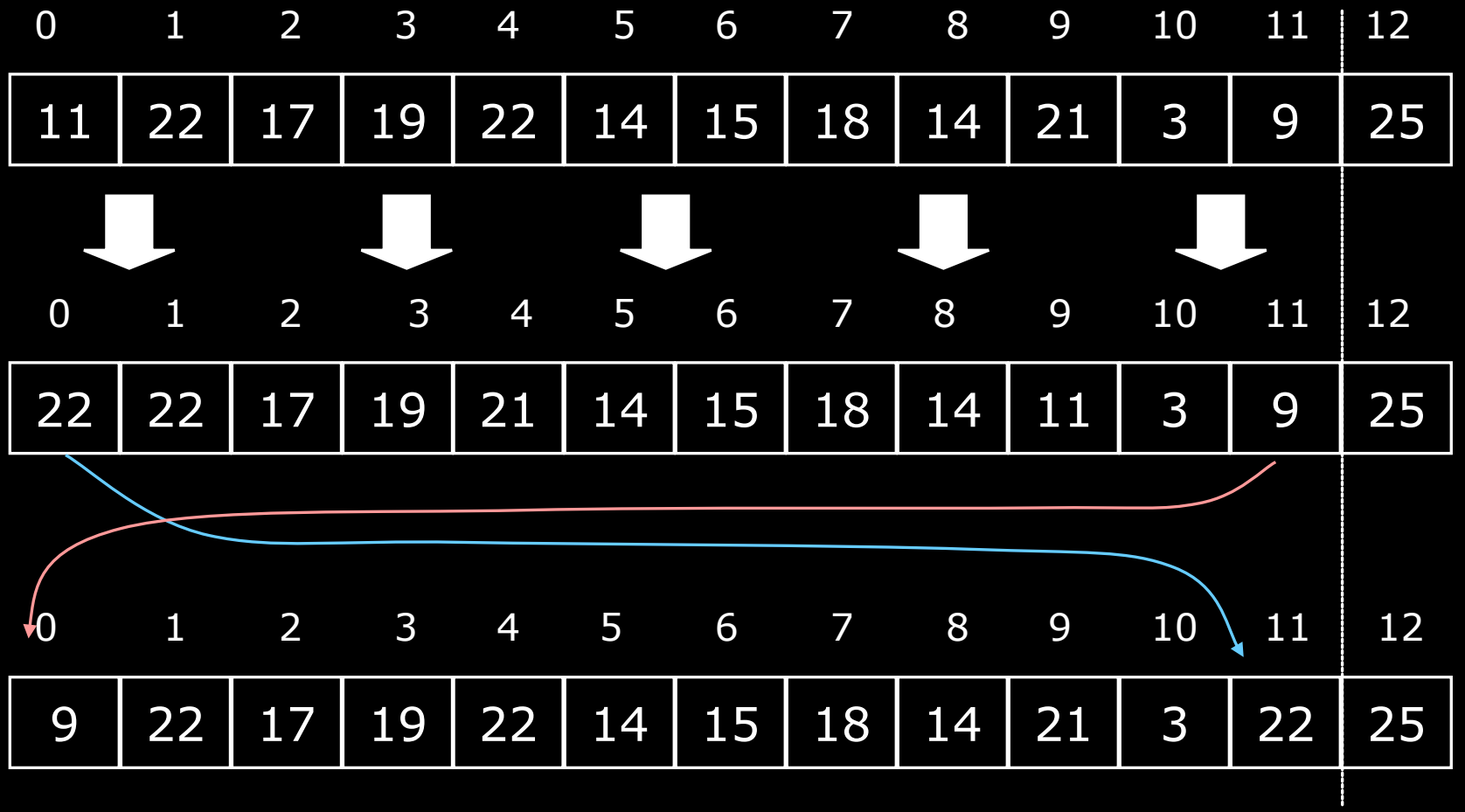
0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Ostráň a nahraď koreň



Koreň je element, ktorý je najviac vľavo
Vymeň ho s elementom najviac vpravo
reHeap...

Po reHeap


0 1 2 3 4 5 6 7 8 9 10 11 12
11 22 17 19 22 14 15 18 14 21 3 9 25
0 1 2 3 4 5 6 7 8 9 10 11 12
22 22 17 19 21 14 15 18 14 11 3 9 25
0 1 2 3 4 5 6 7 8 9 10 11 12
9 22 17 19 22 14 15 18 14 21 3 22 25

Heap: časová zložitosť

urob z poľa heap;

Opakujeme najviac n-krát (pre každý element)

SiftUp od lista ku koreňu: log n

Binárny strom je vyvážený, jeho hĺbka je (log n)

O(n log n)

Heap: časová zložitosť

```
opakuj {
    odstráň a nahraď koreň;  O(1)
    reHeap;  O(log n)
}
```

Priority queue

Heap implementuje abstraktný dátový typ: priority queue

Operácia: nájdi max (min) element

Heap sort: pseudokód

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

1. iterácia: *i* je koreň

l je ľavý potomok

r je pravý potomok

Rekurzia

Heap sort na poli A, uzol *i* má potomkov *l* a *r*

Potreba implementovať funkcie LEFT a RIGHT

Timeline

1730: aproximácia faktoriálu (Stirling)
1945: merge sort (von Neumann)
1955: „umelá inteligencia“ (John McCarthy)
1957: Fortran (John Backus)
1958: Lisp (John McCarthy)
1960: quick sort (Hoare)
1964: heap sort (J. W. J. Williams)
1972: C (Dennis Ritchie)
1983: C++ (Stroustrup)
1995: Java (Gosling)