Sir Charles Antony Richard Hoare
C.A.R. (Tony) Hoare

11.1..1934 Colombo, Ceylon – britský informatik

1956 - Bc klasické štúdiá, Oxford

1968 – profesor informatiky na Královninej unioverizte v Belfaste

1977 – profesor informatiky na Oxforde

1960 – quicksort

Hoarova logika pre dokazovanie správnosti programov

Communicating Sequential Processes (CSP)

Sú dva spôsoby ako navrhovať softvér: jeden je urobiť ho tak jednoduchý, že v ňom zjavne nie sú nedostatky, druhý je urobiť ho tak zložitý, že v ňom nie sú zjavné nedostatky. Prvý spôsob je omnoho zložitejší.

Rýchle usporadúvanie (Quick Sort)

- quicksort alebo usporadúvanie rozdeľovaním je jeden z najrýchlejších známych algoritmov založených na porovnávaní prvkov
- jeho priemerná doba výpočtu je najlepšia zo všetkých podobných algoritmov
- nevýhodou je, že pri nevhodnom usporiadaní vstupných dát môže byť časová aj pamäťová náročnosť omnoho väčšia

Rýchle usporadúvanie (Quick Sort)

```
QUICKSORT(A, p, r)
  if p < r
     then q ← PARTITION(A, p, r)
          QUICKSORT(A, p, q − 1)
          QUICKSORT(A, q + 1, r)
```


rozčlenenie

```
PARTITION(A, p, r)
  x ← A[r]
  i ← p − 1
  for j ← p to r − 1
      do if A[j] ≤ x
            then i ← i + 1
                 exchange A[i] ↔ A[j]
  exchange A[i + 1] ↔ A[r]
  return i + 1
```


rozčlenenie

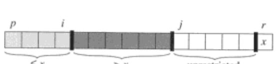


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p\,..\,r]$. The values in $A[p\,..\,i]$ are all less than or equal to x, the values in $A[i+1\,..\,j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j\,..\,r-1]$ can take on any values.

rozčlenenie

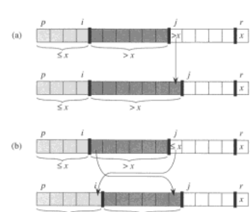


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Hoarova formulácia rozčleňovania

```
HOARE-PARTITION(A, p, r)
x ← A[p]
i ← p − 1
j ← r + 1
while TRUE
     do repeat j ← j − 1
          until A[j] ≤ x
        repeat i ← i + 1
          until A[i] ≥ x
        if i < j
          then exchange A[i] ↔ A[j]
          else return j
```


trochu iná formulácia rozčleňovania

```
Partition(A,left,right):int
 p:=A[left]; l:=left+1; r:=right;
 while l<r do
    while A[l]<p do l:=l+1;
    while A[r]≥p do r:=r-1;
    if l<r then swap(A, l, r)
 A[left]:=A[r]; A[r]:=p;
 return r;
```


príklad rozčleňovania

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6 (l > r)
- swap with pivot: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

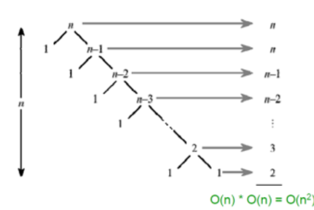
analýza rýchleho usporadúvania

- *najhorší prípad?*
  - rozčlenenie je vždy nevyvážené
- *najlepší prípad?*
  - rozčlenenie je dokonale vyvážené
- *ktorý prípad je častejší?*
  - ten druhý, s prevahou, okrem...
- *je nejaký vstup, ktorý spôsobí najhorší prípad?*
  - áno: usporiadaná postupnosť

analýza rýchleho usporadúvania

- v najhoršom prípade:
  $T(1) = \Theta(1)$
  $T(n) = T(n - 1) + \Theta(n)$
- z čoho vyjde
  $T(n) = \Theta(n^2)$

analýza rýchleho usporadúvania

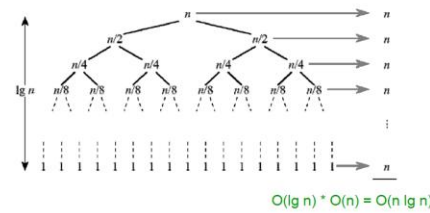


analýza rýchleho usporadúvania

- v najlepšom prípade:
  T(n) = 2T(n/2) + Θ(n)
- z čoho vyjde
  T(n) = Θ(n lg n)

63

analýza rýchleho usporadúvania



64

vylepšenie rýchleho usporadúvania

- naozajstnú záruku dáva r.u. na usporiadanú postupnosť - vtedy je kvadratický O(n^2)
- možnosti vylepšenia:
  - znáhodnenie (poradia) vstupnej postupnosti alebo
  - náhodná voľba pivota
- *v čom je vylepšenie?*
  - zabezpečením, že vstup nebude taký, že spôsobí r.u. vykonávané v kvadratickom čase O(n^2)

65

analýza rýchleho usporadúvania: priemerný prípad

- za predpokladu „náhodného“ vstupu je čas v priemernom prípade omnoho bližší k O(n lg n) než k O(n^2)
- názorné vysvetlenie/príklad:
  - predpokladajme, že rozčlenenie vždy dopadne 9-ku-1 (dosť nevyvážené)!
  - rekurentná rovnica:
    T(n) = T(9n/10) + T(n/10) + n

66

analýza rýchleho usporadúvania: priemerný prípad

- intuitívne sa dá predpokladať, že v skutočnosti r.u. prebehne ako zmes “zlých” a “dobrých” rozčlenení
  - dobré a zlé budú rozložené náhodne v strome rekurzie
  - predpokladajme, že sa bude striedať najlepší (n/2 : n/2) a najhorší prípad (n-1 : 1)
  - *čo sa stane, ak sa zle rozčlení hneď koreňový vrchol a potom sa dobre rozčlení z toho rezultujúci (n-1) vrchol?*

67

analýza rýchleho usporadúvania: priemerný prípad

- intuitívne sa dá predpokladať, že v skutočnosti r.u. prebehne ako zmes “zlých” a “dobrých” rozčlenení
  - dobré a zlé budú rozložené náhodne v strome rekurzie
  - predpokladajme, že sa bude striedať najlepší (n/2 : n/2) a najhorší prípad (n-1 : 1)
  - *čo sa stane, ak sa zle rozčlení hneď koreňový vrchol a potom sa dobre rozčlení z toho rezultujúci (n-1) vrchol?*
    - dostaneme 3 podpostupnosti s veľkosťami 1, (n-1)/2, (n-1)/2
    - celková cena rozčlenení = n + n -1 = 2n -1 = O(n)
    - o nič horšie ako keby sme mali dobré rozčlenenie hneď v koreňovom vrchole!

68

zlepšenie voľbou lepšieho pivota

- zatiaľ sme volili
  - krajný prvok (síce O(1), ale...)
  - jedno, či prvý alebo posledný
- zaručene najlepšia voľba?
  - koľko prvkov vľavo, toľko vpravo od neho
  - porovnaj definíciu mediánu
  - ako určiť medián? usporiadať a zvoliť prvok presne v strede!
- znáhodnenie

69

Iný spôsob voľby pivota a rozčlenenia

```
RANDOMIZED-PARTITION(A, p, r)
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

70

Znáhodnené rýchle usporadúvanie

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
   then q ← RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICKSORT(A, p, q − 1)
        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

71

Varianta

```
QUICKSORT′(A, p, r)
while p < r
   do ▷ Partition and sort left subarray.
      q ← PARTITION(A, p, r)
      QUICKSORT′(A, p, q − 1)
      p ← q + 1
```

72

Rýchle usporadúvanie - zhrnutie

- Základom je rozdelenie postupnosti na dve časti. V jednej časti sú čísla väčšie a v druhej menšie ako zvolená hodnota, ktorá sa nazýva **pivot**
- Je dôležité správne zvoliť pivot, najlepšie tak, aby rozdelené časti boli približne rovnako veľké, čím sa získa najrýchlejšie možné usporiadanie

73

Advanced Data Structures, Cataltepe and Altilar 2005-2007, ITU

Performance of Quick Sort

Worst case: If one of the partitions are always empty.
$T(n)=T(n-1)+T(0)+\Theta(n)= T(n-1)+\Theta(n)=\Theta(n^2)$
Best case: If partitions are always equal sized:
$T(n)<=2T(n/2)+\Theta(n)=O(n\lg n)$
Average case: If all partition sizes are equally likely:
$T(n)=O(n\lg n)$
See the proof in 7.4. on page 155.
RANDOMIZED-PARTITION: Randomly permute the array elements before sorting and hence achieve the average case performance of quick sort.

Week 3: Sorting: Insertion, Merge, Heap, Quick 78

74

Rýchle usporadúvanie (Quick Sort)

```
procedure quickSort(pole, lavy, pravy)
  if lavy < pravy
     pivot := lavy
     for i := lavy + 1 to pravy
          if pole[i] < pole[lavy]
               pivot := pivot + 1
               swap(pole[pivot], pole[i])
          end
     end
     swap(pole[pivot], pole[lavy])
     quickSort(pole, lavy, pivot)
     quickSort(pole, pivot + 1, pravy)
  end
end
```


Porovanie jednotlivých metód

	časová zložitosť
Vkladaním	0(N^2)
Výmenou	0(N^2)
Výberom	0(N^2)
Zlučovaním dvojcestné	0(N log N)
Radixové	0(N log N)
Výpočtom adries	0(N)
Shellovo	0(N $\log^2$ N)
rýchle	0(N log N)

János (John) von Neumann

(December 28, 1903, Budapest – February 8, 1957)

americký matematik, narodený v Rakúsku-Uhorsku.

- matematická štatistika a ekonometria,
- kvantová mechanika,
- ekonómia a teória hier,
- informatika:
  - architektúra počítačov
  - merge sort

Rozdeľuj a panuj – divide et impera

- metóda
  - riešenia problémov
  - navrhovania algoritmov

- ak problém je dostatočne jednoduchý,
- tak ho vyrieš priamo

- ak problém
  - je rozsiahly
  - dá rozdeliť na viacero menších neprekrývajúcich sa podproblémov
- tak
  - **rozdeľ** problém na 2 alebo viac menších podproblémov
  - (**panuj**) použi ten istý postup rekurzívne na riešenie podproblémov
  - **skombinuj** získané riešenia podproblémov do riešenia pôvodného problému

Usporadúvanie zlučovaním (merge sort)

- vychádza z metódy rozdeľuj a panuj, t.j. ľahšie sa usporiada menej položiek ako veľa

- usporadúvanie zlučovaním je opakom usporadúvania rozdeľovaním (quick sort)

- skladá sa z dvoch častí: rozdelenie na usporiadané podpostupnosti a opätovné spájanie

- usporiadané podpostupnosti sa rekurzívne zlučujú do jednej spoločnej usporiadanej postupnosti

Usporadúvanie zlučovaním (merge sort)

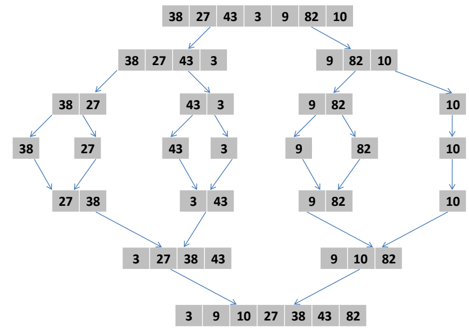
- postupnosť najprv rozdelíme na dve približne rovnako veľké časti (pri nepárnom počte je jedna časť väčšia)
- ďalej sa budeme zaoberať každou z týchto dvoch častí zvlášť, a to takým istým spôsobom, t.j. rozdelíme ich na dve časti
- znovu vezmeme prvú z nich a rozdelíme ju na dve, atď. ... až kým nedostaneme jednoprvkové úseky
- je zrejmé, že jednoprvkové úseky sú vždy usporiadané
- teraz použijeme druhú časť algoritmu: zlúčenie dvoch usporiadaných častí tak, aby aj novovzniknutá časť bola usporiadaná, t.j. dostávame časť s dvoma položkami
- podobne sa rozdelí a zlúči aj druhá časť z rozdelenia a dostávame usporiadanú druhú dvojprvkovú časť
- ten istý algoritmus zlúčenia dvoch usporiadaných častí použijeme aj teraz a dostávame usporiadanú štvorprvkovú časť, atď.
- algoritmus sa bude opakovať, až kým nebude usporiadané celé pole

Usporadúvanie zlučovaním (merge sort)

- Výhody oproti quick sort:
  - stabilný algoritmus usporiadania
  - lepšie možnosť využitia paralelného spracovania
  - sekvenčný prístup k údajom umožňuje pracovať nad veľkým množstvom údajov, uložených na médiach so sekvenčným prístupom, bez nutnosti načítavať tieto údaje do pamäte
  - umožňuje „on-line“ pridávanie ďalších podpostupností (ktoré sa najprv usporiadajú) počas procesu zlučovania.

81

Usporadúvanie zlučovaním (merge sort)



82

Usporadúvanie zlučovaním (merge sort)

- Koncept rekurzívneho algoritmu:
  1. ak je postupnosť dĺžky 0 alebo 1, tak je postupnosť usporiadaná, ak nie, tak rozdeľ neusporiadanú postupnosť na dve podpostupnosti s polovičnou veľkosťou
  2. usporiadaj každú podpostupnosť rekurzívnym aplikovaním tohože algoritmu
  3. zlúč dve usporiadané podpostupnosti do jednej usporiadanej postupnosti.

83

MERGE-SORT(A,p,r)

1. **if** $p < r$
2. **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MERGE-SORT(A,p,q)
4. MERGE-SORT(A,q+1,r)
5. MERGE(A,p,q,r)

na usporiadanie postupnosti zapísanej v A[1..n] sa zavolá MERGE-SORT(A,1,n) (n=length(A))

84

Merge – zlúč

- predpodmienka:
  - A[p..q] a A[q+1..r] sú usporiadané
- popodmienka
  - A[p..r] je usporadané

Merge (A,p,q,r)

$$s \leftarrow q - p + 1; t \leftarrow r - q;$$
$$L \leftarrow A[p..q]; R \leftarrow A[q+1, r]$$
$$L[s+1] \leftarrow \infty; R[t+1] \leftarrow \infty$$
$$A[p..r] \leftarrow MergeArray(L, R)$$

85

MergeArray – zlúč polia

- predpodmienka:
  - L[1..s] a R[1..t] sú usporiadané
- popodmienka
  - MergeArray(L, R) vytvorí jeden usporiadaný vektor A[1..s+t] z prvkov v L a R

A = MergeArray(L,R)
- $L[s+1] \leftarrow \infty; R[t+1] \leftarrow \infty$
- $i \leftarrow 1; j \leftarrow 1$
- **for** $k \leftarrow 1$ **to** $s + t$
  - **do if** $L[i] \le R[j]$
    - **then** $A[k] \leftarrow L[i]; i \leftarrow i+1$
    - **else** $A[k] \leftarrow R[j]; j \leftarrow j+1$

86

Správnosť procedúry MergeArray

Correctness of MergeArray

- Loop-invariant
  - At the start of each iteration of the **for** loop, the subarray $A[1:k-1]$ contains the $k-1$ smallest elements of $L[1:s+1]$ and $R[1:t+1]$ in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A

87

Správnosť procedúry MergeArray

Inductive Proof of Correctness

Initialization: (the invariant is true at beginning)

prior to the first iteration of the loop, we have $k = 1$, so that $A[1,k-1]$ is empty. This empty subarray contains $k-1 = 0$ smallest elements of L and R and since $i = j = 1$, $L[i]$ and $R[j]$ are the smallest element of their arrays that have not been copied back to A.

88

Správnosť procedúry MergeArray

Inductive Proof of Correctness

Maintenance: (the invariant is true after each iteration)

WLOG: assume $L[i] <= R[j]$, the $L[i]$ is the smallest element not yet copied back to A. Hence after copy $L[i]$ to $A[k]$, the subarray $A[1..k]$ contains the k smallest elements. Increasing k and i by 1 reestablishes the loop invariant for the next iteration.

89

Správnosť procedúry MergeArray

Inductive Proof of Correctness

Termination: (loop invariant implies correctness)

At termination we have k = s+t + 1, by the loop invariant, we have A contains the k-1 (s+t) smallest elements of L and R in sorted order.

90

Časová výpočtová zložitosť MergeArray

- v každom kroku cyklu sa vykoná:
  - 1 porovnanie
  - 1 priradenie (skopírovanie jedného prvku do A)
  - 2 inkrementácie (*k* a *i* alebo *j*)
- spolu v každom kroku cyklu 4 operácie
- celkovo čas 4.(s+t), čiže O(n)

91

časová zložitosť rozdeľuj a panuj

- opisuje rekurentná rovnosť
- predpokladajme, že $T(n)$ je čas riešenia problému rozsahu n.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \le n_c \\ aT(n/b)+D(n)+C(n) & \text{if } n > n_c \end{cases}$

  kde a: počet podproblémov

  n/b: veľkosť každého podproblému

  $D(n)$: cena operácie rozdelenia

  $C(n)$: cena operácie kombinovania (spájania)

92

časová zložitosť rozdeľuj a panuj

$$T(n)=\begin{cases}\text{solving\_trivial\_problem} & \text{if } n=1\\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor})+\text{dividing}+\text{combining} & \text{if } n>1\end{cases}$$

$$T(n)=\begin{cases}1 & \text{if } n=1\\ 2T(n/2)+n & \text{if } n>1\end{cases}$$

93

analýza MERGE-SORT

- **Divide**: $D(n) = \Theta(1)$
- **Impera**: a=2,b=2, so $2T(n/2)$
- **Skombinuj**: $C(n) = \Theta(n)$
- $T(n) = \begin{cases}\Theta(1) & \text{if } n=1\\ 2T(n/2)+\Theta(n) & \text{if } n>1\end{cases}$
- $T(n) = \begin{cases}c & \text{if } n=1\\ 2T(n/2)+cn & \text{if } n>1\end{cases}$

94

Časová výpočtová zložitosť usporadúvania zlučovaním

Analysis of Merge Sort

$T(n)$	MERGE-SORT $A[1\,..\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
$\Theta(n)$	3. *"Merge"* the 2 sorted lists

Abuse

Sloppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.

95

rekurentná rovnosť pre T(n)

Recurrence for Merge Sort

$$T(n)=\begin{cases}\Theta(1) \text{ if } n=1;\\ 2T(n/2)+\Theta(n) \text{ if } n>1.\end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.

96

rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

97

rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

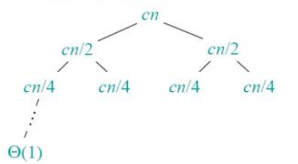
$T(n)$

98

rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

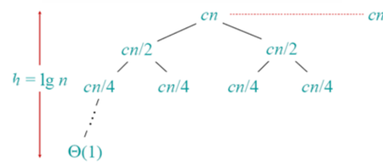
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

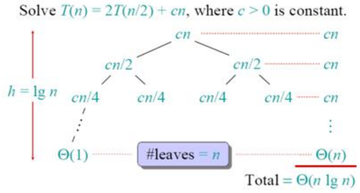
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



rekurentná rovnosť pre T(n) - riešenie

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

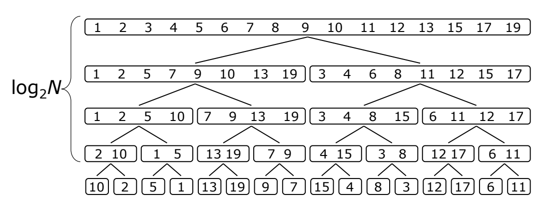
$$\begin{aligned}
T(n) &= 2T(n/2) + n && \text{substitute} \\
&= 2\left(2T(n/4) + n/2\right) + n && \text{expand} \\
&= 2^2 T(n/4) + 2n && \text{substitute} \\
&= 2^2 (2T(n/8) + n/4) + 2n && \text{expand} \\
&= 2^3 T(n/8) + 3n && \text{observe the pattern} \\
T(n) &= 2^i T(n/2^i) + in && \textit{Let } 2^i = n,\ i = \lg n \\
&= 2^{\lg n} T(n/n) + n \lg n = n + n \lg n
\end{aligned}$$

porovnanie usporadúvania vkladaním a zlučovaním

Insertion and Merge Sort

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!

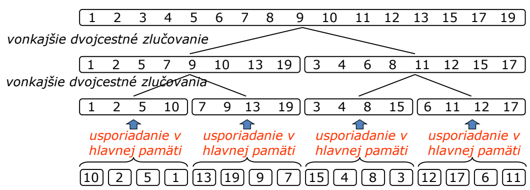
strom rekurzie



- na každej úrovni: zlúč usporiadané úseky (podpostupnosti) veľkosti *x* do úsekov veľkosti *2x*, zníž počet úsekov na polovicu.
- ako by sa tento postup dal použiť na údaje zapísané v súbore vo vedľajšej pamäti?

usporadúvanie zlučovaním vo vonkajšej pamäti

- myšlienka: zväčšiť veľkosť pôvodných úsekov
  - veľkosť pôvodného úseku podľa veľkosti dostupnej časti hlavnej pamäti (*M* miest)

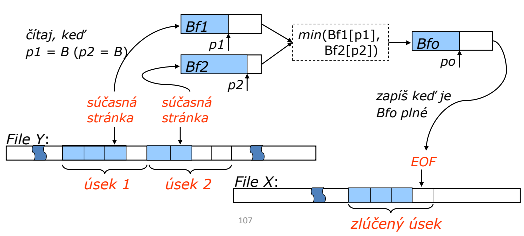


usporadúvanie zlučovaním vo vonkajšej pamäti

- vstupný súbor *X*, prázdny súbor Y
- *fáza* 1: repeat until eof(*X*):
  - prečítaj ďalších *M* prvkov z *X*
  - usporiadaj ich v hlavnej pamäti
  - zapíš ich na koniec súboru *Y*
- *fáza* 2: while not empty(*Y*):
  - vyprázdni *X*
  - *MergeAllRuns*(*Y*, *X*)
  - *súbor X* sa premenuje na *Y*, *Y* sa premenuje na *X*

zlučovanie pri vonkajšom usporadúvaní

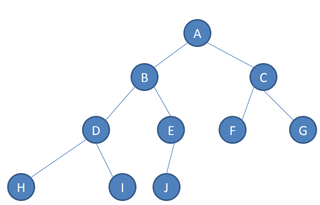
- *MergeAllRuns*(*Y*, *X*): repeat until eof(*Y*):
  - vykonaj *TwowayMerge*, aby sa zlúčili nasledujúce dva úseky z Y do jedného, ktorý sa zapíše na koniec *X*
- *TwowayMerge*: používa 3 vektory v hlavnej pamäti veľkosti B



Usporadúvanie haldou (Heap Sort)

- Pri usporadúvaní využijeme špeciálny pojem **halda** - je to dátová štruktúra, ktorá:
  - má tvar "skoro" úplného binárneho stromu (len na poslednej úrovni binárneho stromu môžu chýbať synovia (vrcholov predposlednej úrovne) a to tak, že ak chýba nejaký syn, tak budú chýbať aj všetci synovia vpravo na najnižšej úrovni
  - pre všetky vrcholy stromu platí, že otec má väčšiu (alebo rovnú) hodnotu ako jeho synovia - ak existujú
  - haldu budeme reprezentovať v jednorozmernom poli indexovanom od 0 tak, že koreň stromu je na indexe 0 a i-ty vrchol má synov na indexoch **2*i+1** a **2*i+2**

Usporadúvanie haldou



	A	B	C	D	E	F	G	H	I	J	
0	1	2	3	4	5	6	7	8	9	10	11

Usporadúvanie haldou

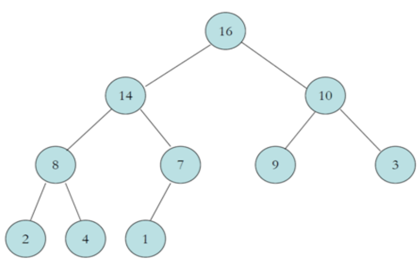
- usporadúva prvky pomocou dátovej štruktúry binárna halda
- predstavuje efektívnejšiu verziu usporadúvania výberom
- najväčší (resp. najmenší) prvok sa vyberá z koreňa max-haldy (resp. min-haldy)
- max-halda je strom, pre ktorý platí, že každý potomok v strome má menšiu hodnotu ako jeho rodič (min-halda naopak)

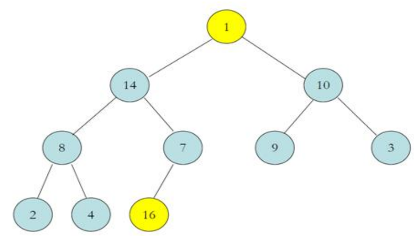
Usporadúvanie haldou

- samotný algoritmus má dve fázy:
  - vytvorenie haldy
  - v halde je koreň (t.j. prvý prvok poľa) najväčší prvok zo všetkých, jeho výmena s posledným prvkom (ešte neusporiadaného) poľa a nové „vyhaldovanie", t.j. oprava haldy
- zakaždým pracujeme s o 1 kratším poľom - haldou -> na jeho konci sa postupne sústreďujú najväčšie prvky

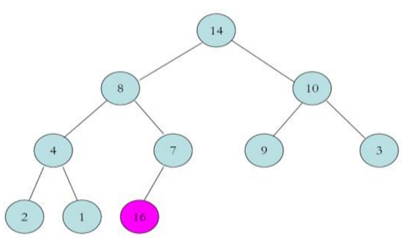
Heapsort

HEAPSORT (A)
    BUILD-MAX-HEAP(A)
    for $i \leftarrow$ *length*[A] **downto** 2
        do exchange $A[1]$ **with** $A[i]$
            heap-size[A] $\leftarrow$ *heap-size*[A] - 1
            MAX-HEAPIFY(A, 1)

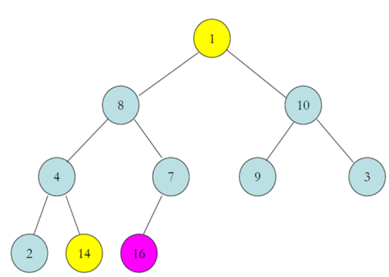


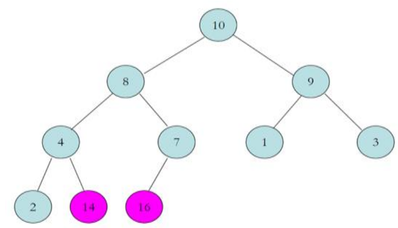


Swap $A[1] \leftrightarrow A[i]$



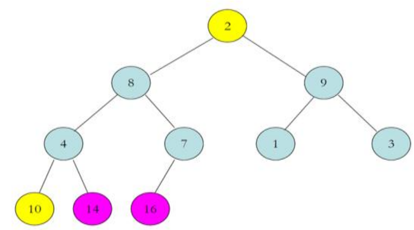
heap-size $\leftarrow$ heap-size – 1
MAX-HEAPIFY(A, 1)





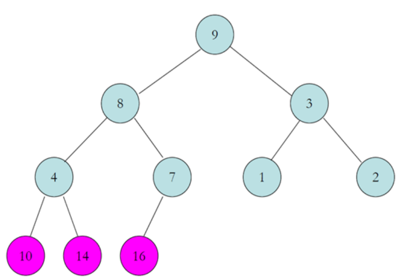
heap-size ← heap-size – 1
MAX-HEAPIFY(A, 1)

117

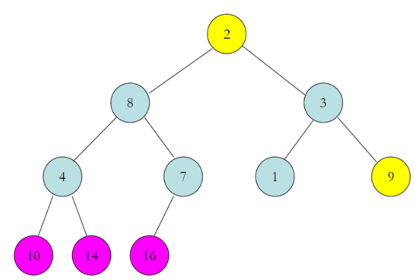


Swap $A[1] \leftrightarrow A[i]$

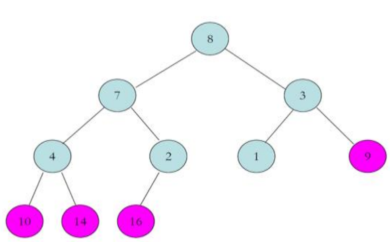
118



119



120

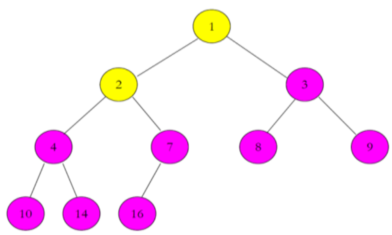


heap-size ← heap-size – 1
MAX-HEAPIFY(A, 1)

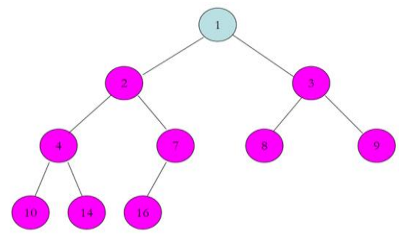
121

- ....

122



Swap $A[1] \leftrightarrow A[i]$



heap-size $\leftarrow$ heap-size $-$ 1
MAX-HEAPIFY(A, 1)

Usporadúvanie haldou (1)

```
procedure posun(var i: Integer; m:
  Integer)
begin
  // ak existuje syn, nastaví sa na väčšieho z nich
  if 2*i+1 <= m then // ak aspoň jeden syn existuje
  begin
     i := 2*i+1; // i je ľavý syn
     if (i < m) and (p[i+1] > p[i]) then
          i := i+1; // i je teraz už pravý syn - bol väčší
     Inc(pocet);
  end;
end;
```


Usporadúvanie haldou (2)

```
procedure uprac_haldu(k, m: Integer)
var
  i: Integer;
begin
  // predpokladáme, že od k+1 do m to už halda je - pridáme k-ty
  i := k;
  posun(i, m);
  while p[k] < p[i] do
  begin
     vymen(k, i);
     k := i;
     posun(i, m); // i je nový väčší syn
     Inc(pocet);
  end;
  Inc(pocet);
end;
```


Usporadúvanie haldou (3)

```
procedure vytvor_haldu
   var i: Integer;
begin
   for i := (N-1) div 2 downto 0 do
        uprac_haldu(i, N-1);
end;

Procedure HEAPSORT
var posledny: Integer;
begin
   vytvor_haldu;
   posledny := N-1; // ber postupne všetky prvky

   while posledny > 0 do
   begin
        vymen(0, posledny); // vymeň koreň s posledným prvkom
        Dec(posledny); // zmenši rozmer stromu
        uprac_haldu(0, posledny); // oprav strom - koreň je teraz asi zlý
   end;
end;
```


Usporadúvanie haldou

- procedúra uprac_haldu vytvára haldu v poli medzi zadanými dvoma indexmi poľa
- po jej skončení je časť poľa K až M haldou, pričom procedúra predpokladá, že časť K+1 až M je halda, ale tým, že sme pridali prvok na miesto K, mohla sa halda K až M pokaziť
- preto je potrebné pole znovu „vyhaldovať", t.j. zabezpečiť, aby aj pre K platilo, že má oboch synov menších ako on sám (ak to tak nie je, treba ho zrejme vymeniť s väčším zo synov a postup opakovať pre tohto syna a príslušný podstrom)
- pomocná procedúra posun posúva prvý parameter na väčšieho syna

Usporadúvanie haldou - zložitosť

- heap sort má rovnaké časové zložitosti ako merge sort, t.j. garantuje zložitosť **O(n log n)**
- výhodou oproti merge sort je tzv. in-line pamäťová zložitosť – O(1), t.j. nepotrebuje dodatočnú pamäť, pracuje priamo nad pamäťou vstupnej postupnosti – merge sort O(n)
- nevýhody oproti merge sort:
  - Heap sort vyžaduje priamy prístup k údajom
  - Sekvenčný prístup merge sort-u môže lepšie využiť potenciál pamäte cache
  - Heap sort sa nedá paralelizovať
  - Heap sort nie je stabilný

129