

Vyhľadávanie

Vyhľadávanie

Vstupy:

- N prvkov identifikovateľných kľúčom
- kľúč K, ktorý charakterizuje prvok, ktorý chceme nájsť

Výstupy:

- Úspech : prvok sa v zadanej množine našiel
- Neúspech: prvok sa nenašiel

Rozdelenie vyhľadávaní

- vnútorné, vonkajšie
- statické, dynamické
- uporiadaná, neusporiadaná postupnosť
- lineárne, binárne, pomocou rozptylovej funkcie

Jednoduché vyhľadávanie

Vstup: pole prvkov, kľúč

Výstup: pozícia nájdeného prvku v poli (ak sa prvok nenájde, tak sa vráti 0)

```
Procedure SEARCH(POST:pole, K: kľúč, var KDE: integer )
VAR NAJDENY : BOOLEAN;
I : 1..VELKOST;
BEGIN
NAJDENY := FALSE;
I := 1;
WHILE((I <= VELKOST) AND (NOT NAJDENY)) DO
IF K = POST[I].KLUC THEN
BEGIN
KDE := I;
NAJDENY := TRUE;
END
ELSE
I++;
IF NOT NAJDENY THEN
KDE := 0;
END;
```

Binárne vyhľadávanie

- Vstupom je usporiadané pole.
- Algoritmus porovná prvok nachádzajúci sa v strede poľa so zadaným kľúčom. Ak sa zhoduje, vráti jeho poziciu. Ak sa nezhoduje, rozdelí pole na 2 polovice a pokračuje rovnakým hľadáním v tej polovici v ktorej sa prvok nachádza.

- Rekurzívna verzia algoritmu:

```
BinarySearch(A[0..N-1], value, low, high)
{
    if (high < low)
        return not_found
    mid = (low + high) / 2
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid
}
```

Binárne vyhľadávanie

- Nájdi prvok 92

11 14 16 27 31 35 39 39 43 49 50 58 66 72 74 80 85 92 92 97 97 97

Usporadúvanie

Usporadúvanie

- Postupnosť: p1, p2, p3...pN

- Položka:

- Pi
- Klúč Ki (hodnota položky)

Usporiadanie: binárna relácia

Úlohou je preusporiadať všetky položky postupnosti tak, aby platilo:

K1 <= K2 <= K3 <= ... <= Kn

Stabilný algoritmus

- Usporadúvací algoritmus je stabilný, ak vždy zachová originálne poradie elementov s rovnakými klúčmi
- Ak elementy s rovnakými klúčmi sú neodlísiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu (napr. ak klúcom je samotný element)
- Zachovať originálne poradie elementov je dôležité napr. pri viacnásobnom usporiadaní – najprv podľa predziska a potom podľa mena.

Stabilný algoritmus

- Každý nestabilný algoritmus sa dá implementovať ako stabilný tým, že sa zapamäta originálne poradie elementov a pri zhodných klúčoch sa berie do úvahy toto poradie
- Viacnásobné usporiadanie je možné obistiť vytvorením jedného klúča, ktorý je zložený z primárneho, sekundárneho, atď. klúča usporiadania
 - Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť.

Stabilný algoritmus

- Priklad – dvojice (klúč, element):
 - (4, 5) (2, 7) (2, 3) (5, 6)
- Dve možné usporiadania:
 - (2, 7) (2, 3) (4, 5) (5, 6) – zachované poradie elementov s klúčmi 2 – stabilné usporiadanie
 - (2, 3) (2, 7) (4, 5) (5, 6) – zmenené poradie elementov s klúčmi 2 – nestabilné usporiadanie
- Priklad na viacnásobné usporiadanie – dvojice (klúč 1, klúč 2):
 - (4, 5) (2, 7) (2, 3) (4, 6)
- Usporiadanie najprv podľa klúča 2, potom podľa klúča 1:
 - (2, 3) (4, 5) (4, 6) (2, 7) – podľa klúča 2
 - (2, 3) (2, 7) (4, 5) (4, 6) – podľa klúča 1
- Usporiadanie najprv podľa klúča 1, potom podľa klúča 2:
 - (2, 7) (2, 3) (4, 5) (4, 6) – podľa klúča 1
 - (2, 3) (4, 5) (4, 6) (2, 7) – podľa klúča 2 – narušené poradie
- Pre zachovanie stability viacnásobného usporadúvania je potrebné usporadúvať postupne podľa klúčov so zvyšujúcou sa prioritou.

Porovnávací algoritmus

- usporadúvací algoritmus, ktorý prechádza vstupné klúče a na základe operácie porovnávania rozhoduje, ktorý z dvoch elementov sa má v usporiadanom poli objaviť ako prvý.
- Operácia porovnávania musí mať tieto vlastnosti:
 - Ak $a \leq b$ a $b \leq c$, tak potom $a \leq c$
 - Pre všetky a a b , bud' $a \leq b$ alebo $b \leq a$
- Základným limitom je dolné ohrazenie počtu porovnávania $\Omega(n \log n)$, ktoré je potrebné na usporiadanie postupnosti. Preto aj tie najlepšie algoritmy usporadúvania založené na porovnávaní, majú priemernú časovú zložitosť $O(n \log n)$ – na rozdiel od neporovnávacích algoritmov, kde sa môže dosiahnuť časová zložitosť aj $O(n)$.

Výhody porovnávacích algoritmov

- Použiteľné pre rôzne dátové typy
- Jednoduchá implementácia porovnávania n-tíc v lexikografickej postupnosti
- Reverzná funkcia porovnávania = reverzne usporiadaná postupnosť

Najznámejšie algoritmy usporadúvania

- založené na porovnávaní:
 - usporadúvanie výberom,
 - vkladaním,
 - výmenou,
 - zlúčovaním,
 - quicksort,
 - heapsort, ...
- neporovnávacie algoritmy:
 - radix sort,
 - counting sort,
 - bucket sort, ...

Usporadúvanie vkladaním

- algoritmus, ktorý realizuje usporadúvanie priamym vkladaním.
- vychádza z predpokladu, že do už usporiadanej postupnosti sa na správne miesto vloží ďalší prvok.
- ak sa miesto, na ktoré sa nový prvok vkladá, zistíuje binárny vyhľadávaním, hovoríme o usporadúvaní binárnym vkladaním

Usporiadanie vkladaním

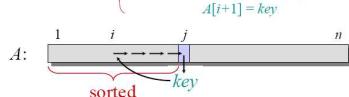
- Príklad:
 - usporiadajte vkladaním pole **6 4 5 2 3 1 7**

- 1. krok **6 | 4 5 2 3 1 7 → 4 6 | 5 2 3 1 7**
- 2. krok **4 6 | 5 2 3 1 7 → 4 5 6 | 2 3 1 7**
- 3. krok **4 5 6 | 2 3 1 7 → 2 4 5 6 | 3 1 7**
- 4. krok **2 4 5 6 | 3 1 7 → 2 3 4 5 6 | 1 7**
- 5. krok **2 3 4 5 6 | 1 7 → 1 2 3 4 5 6 | 7**
- 6. krok **1 2 3 4 5 6 | 7 → 1 2 3 4 5 6 7 |**
- 7. krok **1 2 3 4 5 6 7 | → 1 2 3 4 5 6 7 |**

Usporiadanie vkladaním

- Časová zloženosť závisí od vstupného poľa
 - pre takmer usporiadané vstupné pole algoritmus prebehne rýchlo – vtedy sa akoby vymieňali len vymieňali dva susedné prvky (najpravejší z usporiadanej časti s najľavejším z neusporiadanej časti) a nedochádza tak k posunu ostatných prvkov.

“pseudocode” {
INERTION-SORT (A, n) ▷ $A[1 \dots n]$
for $j \leftarrow 2$ to n
do $key \leftarrow A[j]$
 $i \leftarrow j - 1$
while $i > 0$ and $A[i] > key$
do $A[i+1] \leftarrow A[i]$
 $i \leftarrow i - 1$
 $A[i+1] = key$



Usporiadanie vkladaním

```

Procedure InsertionSort(var A: pole);
Var i, j, x : Integer;

BEGIN
  for i := 2 to Dlzka(A) do begin
    x := A[i];
    a[0] := x;
    j := i - 1;
    while x < A[j] do
      begin
        a[j+1] := a[j];
        j := j - 1;
      end;
    a[j+1] := x;
  end
END.

```

Usporadúvanie výmenou - bublinkové (bubble)

- bubble sort usporadúva prvky priamou výmenou
 - je implementačne jednoduchý, ale neefektívny
 - pri usporadúvaní porovnáva dva susedné prvky a ak nie sú v správnom poradí, vymenia sa
 - procedúra sa opakuje, až kým nie sú potrebné žiadne výmeny

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
 - na začiatku sme mali pole **5 1 4 2 8**
 - 3. fáza
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - na konci máme usporiadane pole **1 2 4 5 8**

Insertion Sort Running Time

- *Best case:* $\Theta(n)$, the inner loop is not executed at all

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^n \Theta(j) = \Theta(n^2) \quad [\text{arithmetic series}]$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^n \Theta(j/2) = \Theta(n^2)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n .
 - Not at all, for large n .

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
 - Usporiadajte bublinkovou výmenou pole **5 1 4 2 8**
 - 1. fáza
 - » **5 1 4 2 8** → **1 5 4 2 8**
 - » **1 5 4 2 8** → **1 4 5 2 8**
 - » **1 4 5 2 8** → **1 4 2 5 8**
 - » **1 4 2 5 8** → **1 4 2 5 8**
 - 2. fáza
 - » **1 4 2 5 8** → **1 4 2 5 8**
 - » **1 4 2 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
 - na začiatku sme mali pole **5 1 4 2 8**
 - 3. fáza
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - » **1 2 4 5 8** → **1 2 4 5 8**
 - na konci máme usporiadane pole **1 2 4 5 8**

Bublinkové usporadúvanie

```

Procedure BubbleSort(var A: pole)
Var i, j, t : integer;
Begin
  for i := Dlzka(A) downto 1 do
    for j := 1 to Dlzka(A)-1 do
      if A[j] > A[j+1] then
        begin
          T := A[j];
          A[j] := A[j+1];
          A[j+1] := T;
        end;
  end;
End.

```

Usporadúvanie výberom

- algoritmus realizuje usporadúvanie priamym výberom
- vychádza z prepočtu, že najmenší prvok môžeme zaradiť priamo na začiatok vstupného poľa, najmenší prvok zo zvyšku poľa zase na jeho začiatok atď.
- podľa toho, či usporadúva prvky vzostupne/zostupne, sa môže označovať ako MinSort/MaxSort

Usporadúvanie výberom

- Príklad:
 - usporiadajte výberom pole **6 4 5 2 3 1 7**
- 1. krok | 6 4 5 2 3 1 7 -> 1 | 6 4 5 2 3 7
• 2. krok 1 | 6 4 5 2 3 7 -> 1 2 | 6 4 5 3 7
• 3. krok 1 2 | 6 4 5 3 7 -> 1 2 3 | 6 4 5 7
• 4. krok 1 2 3 | 6 4 5 7 -> 1 2 3 4 | 6 5 7
• 5. krok 1 2 3 4 | 6 5 7 -> 1 2 3 4 5 | 6 7
• 6. krok 1 2 3 4 5 | 6 7 -> 1 2 3 4 5 6 | 7
• 7. krok 1 2 3 4 5 6 | 7 -> 1 2 3 4 5 6 7
- Miera usporiadanosť vstupného poľa nemá vplyv na časovú zložitosť – vždy sa vykoná maximálny počet krokov.

Usporadúvanie výberom

```
procedure SelectionSort(var A: pole)
var i, j, t : integer;
begin
    for i := 1 to Dlzka(A) - 1 do
        for j := Dlzka(A) downto i+1 do
            if A[i] > A[j] then
                begin
                    T := A[i];
                    A[i] := A[j];
                    A[j] := T;
                end;
end;
```

Shellovo usporadúvanie

- algoritmus, ktorý realizuje usporadúvanie priamym vkladaním so zmenšovaním kroku
- je to vlastne zlepšenie usporadúvania vkladaním a bublinkového
- táto metóda je jedna z najrýchlejších pre usporiadanie menších postupností (menej ako 1000 prvkov)

Shellovo usporadúvanie

- Príklad, krokovanie $n = n/2$:
 - Usporiadajte pole **6 4 5 2 8 3 1 7** pomocou algoritmu shell sort, $n = 8/2 = 4$
 - 1. krok, krokovanie 4, vyznačené čísla sa usporiadajú vkladaním
 - » **6 4 5 2 8 3 1 7 -> 6 4 5 2 8 3 1 7**
 - » **6 4 5 2 8 3 1 7 -> 6 3 5 2 8 4 1 7**
 - » **6 3 5 2 8 4 1 7 -> 6 3 1 2 8 4 5 7**
 - » **6 3 1 2 8 4 5 7 -> 6 3 1 2 8 4 5 7**
 - 2. krok, krokovanie 2
 - » **6 3 1 2 8 4 5 7 -> 5 3 5 2 6 4 8 7**
 - » **5 3 5 2 6 4 8 7 -> 1 2 5 3 6 4 8 7**
 - 3. krok, krokovanie 1
 - » **1 2 5 3 6 4 8 7 -> 1 2 3 4 5 6 7 8**

Shellovo usporadúvanie

```
procedure ShellSort(var f: pole)
var i, j, h, v, N : integer;
begin
    N := Dlzka(f);
    h := 1;
    repeat
        h := (3 * h) + 1;
    until h > N;

    repeat
        h := (h div 3);
        for i := (h + 1) to N do begin
            v := f[i];
            j := i;
            while ( (j > h) and ( f[j-h] > v ) ) do begin
                f[j] := f[j - h];
                dec(j, h);
            end;
            f[j] := v;
        end;
    until h = 1;
end
```

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 divide-and-conquer algorithm
 - Divide-the-cores of quicksort!
 - Pick an element, called a pivot, from the array.
 - Reorder the array so that all elements which are less than the pivot come before the pivot, and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
 - Conquer
 - Recursively sort the 2 subarrays
 - Combine
 - Interval since sorting is done in place
 - Characteristics
 - Divide-and-conquerature is like mergesort, but it does not require additional array
 - It sorts in-place
 - Very practical, average performance $O(n \log n)$ (with small constant factors), but worstcase $O(n^2)$

Rýchle usporadúvanie (Quick Sort)

- Partitioning: Key Step in Quicksort
 - We choose some (any) element p in the array as a pivot
 - We then partition the array into three parts based on pivot p :
 - Left part, pivot itself, and right part
 - Partition returns the final position/index of p
- Then, Quicksort will be recursively executed on both left part and right part
 - Quicksort(A, l, r)
 - If $l < r$ then
 - $i := \text{Partition}(A, l, r)$
 - Quicksort($A, l, i-1$)
 - Quicksort($A, i+1, r$)

Rýchle usporadúvanie (Quick Sort)

```
QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2    then  $q \leftarrow \text{PARTITION}(A, p, r)$ 
3    QUICKSORT( $A, p, q - 1$ )
4    QUICKSORT( $A, q + 1, r$ )
```

Partition Algorithm

- Choose an array value (say, the first) to use as the pivot
 - Starting from the left end, find the first element that is greater than or equal to the pivot
 - Searching backward from the right end, find the first element that is less than the pivot
 - Interchange (swap) these two elements
 - Repeat, searching from where we left off, until done
- ```
Partition($A, \text{left}, \text{right}$):int
p:= $A[\text{left}]$; l:= $\text{left}+1$; r:= right ;
while l<r do
 while $A[l] < p$ do l:=l+1;
 while $A[r] \geq p$ do r:=r-1;
 if l < r then swap(A, l, r);
 A[left]:=A[r]; A[r]:=p;
return r;
```

33

## Iná formulácia rozčleňovania

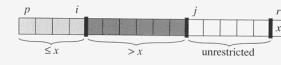
```
HOARE-PARTITION(A, p, r)
1 $x \leftarrow A[p]$
2 $i \leftarrow p - 1$
3 $j \leftarrow r + 1$
4 while TRUE
5 do repeat $j \leftarrow j - 1$
6 until $A[j] \leq x$
7 repeat $i \leftarrow i + 1$
8 until $A[i] \geq x$
9 if $i < j$
10 then exchange $A[i] \leftrightarrow A[j]$
11 else return j
```

## Iný spôsob rozčlenenia

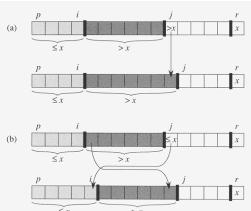
```

PARTITION(A, p, r)
1 $x \leftarrow A[r]$
2 $i \leftarrow p - 1$
3 for $j \leftarrow p$ to $r - 1$
4 do if $A[j] \leq x$
5 then $i \leftarrow i + 1$
6 exchange $A[i] \leftrightarrow A[j]$
7 exchange $A[i + 1] \leftrightarrow A[r]$
8 return $i + 1$

```



**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.



**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  $A[j] \leq x$ , index  $i$  is incremented,  $A[i]$  and  $A[j]$  are swapped, and then  $j$  is incremented. Again, the loop invariant is maintained.

### Example of Partitioning

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

34

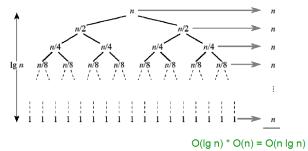
### Best Case of Quicksort

- Suppose each partition operation divides the array of size  $n$  nearly in half
- Then the depth of the recursion in  $\log_2 n$ 
  - That's how many times we can halve  $n$  until we get 1s
- At each level of the recursion, all the partitions at that level do work that is linear in  $n$ 
  - Each partition is linear over its sub-array
  - All the partitions at one level cover the array
- Hence in the best case, quicksort has time complexity  $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Average case also has this complexity
  - Detail omitted here in this course

35

### Best Case Partitioning

- If we are lucky, Partition splits the array evenly



$$O(\lg n) * O(n) = O(n \lg n)$$

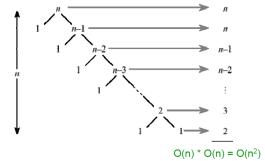
36

### Worst Case of Quicksort

- In the worst case, partitioning always divides the size  $n$  array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length  $n-1$  part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length  $n-1$  part requires (in the worst case) recurring to depth  $n-1$ 
  - In the worst case, recursion may be  $n$  levels deep
- But the partitioning work done at each level is still  $\Theta(n)$
- So worst case for Quicksort is  $\Theta(n) * \Theta(n) = \Theta(n^2)$
- When does this happen?
  - When the array is sorted to begin with!

37

### Worst Case Partitioning



$\Theta(n) * \Theta(n) = \Theta(n^2)$

38

### Picking a Better Pivot

- Before, we picked the *first* element of each sub-array to use as a pivot
  - If the array is already sorted, this results in  $\Theta(n^2)$  behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quicksort (guaranteed  $\Theta(n \log n)$ ) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a median
  - Half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to sort the array and pick the value in the middle ( $\Theta(n \log n)$ )
    - Ironically
    - Random pivot
      - Randomized algorithm of partitioning

39

### Iný spôsob voľby pivota a rozčlenenia

RANDOMIZED-PARTITION( $A, p, r$ )

- $i \leftarrow \text{RANDOM}(p, r)$
- exchange  $A[r] \leftrightarrow A[i]$
- return** PARTITION( $A, p, r$ )

### Znáhodnené rýchle usporadúvanie

```
RANDOMIZED-QUICKSORT(A, p, r)
1 if $p < r$
2 then $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$
3 RANDOMIZED-QUICKSORT($A, p, q - 1$)
4 RANDOMIZED-QUICKSORT($A, q + 1, r$)
```

### Varianta

QUICKSORT'( $A, p, r$ )

- while**  $p < r$
- do** Partition and sort left subarray.
- $q \leftarrow \text{PARTITION}(A, p, r)$
- QUICKSORT'( $A, p, q - 1$ )
- $p \leftarrow q + 1$



## Porovanie jednotlivých metód

*časová zložitosť*

|                       |               |
|-----------------------|---------------|
| Vkladaním             | $O(N^2)$      |
| Výmenou               | $O(N^2)$      |
| Výberom               | $O(N^2)$      |
| Zlučovaním dvojcestné | $O(N \log N)$ |
| Radixové              | $O(N \log N)$ |
| Výpočtom adres        | $O(N)$        |
| Shellovo              | $O(N \log N)$ |
| QuickSort             | $O(N \log N)$ |

## Rozdeľuj a panuj – divide et impera

### Divide and Conquer



- *Divide-and-conquer* is an important algorithm design technique for large-size problems.
- If the problem size is small enough to solve it in a straightforward manner, solve it.
- Otherwise
  - **Divide**
    - Divide the problem into two or more *disjoint* sub-problems
  - **Conquer**
    - Use divide-and-conquer recursively to solve the sub-problems
  - **Combine**
    - Take the solutions to the sub-problems and combine these solutions into a solution for the original problem

3

## Usporadúvanie zlučovaním (merge sort)

### Merge Sort



- Problem: Sort an array  $A$  into non-descending order.
- **Divide**
  - If  $A$  has at least two elements (nothing needs to be done if  $A$  has zero or one elements), remove all the elements from  $A$  and put them into two arrays,  $A_1$  and  $A_2$ , each containing about half of the elements of  $S$  (i.e.,  $A_1$  contains the first  $\lceil n/2 \rceil$  elements and  $A_2$  contains the remaining  $\lfloor n/2 \rfloor$  elements).
- **Conquer**
  - Sort arrays  $A_1$  and  $A_2$  using Merge Sort.
- **Combine**
  - Put back the elements into  $A$ , by merging the sorted arrays  $A_1$  and  $A_2$  into one sorted array

4

## 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 usporiadane podpostupnosti a opäťovné spájanie
- usporiadane podpostupnosti sa rekúrzívne zlučujú do jednej spoločnej usporiadanej postupnosti

## Usporadúvanie zlučovaním (merge sort)

### Merge Sort Algorithm



```

Merge-Sort(A, p, r)
 if p < r then
 q := ⌊(p+r)/2⌋
 Merge-Sort(A, p, q)
 Merge-Sort(A, q+1, r)
 Merge(A, p, q, r)

Merge(A, p, q, r)
 Take the smallest of the two topmost elements of
 arrays A[p..q] and A[q+1..r] and put it into an
 additional array. Repeat this, until both arrays are
 empty. Copy the additional array into A[p..r].

```

**Merge(A, p, q, r)**  
Merge two sorted arrays into one.  
Time complexity:  $O(|A| + |B|)$ .

5

## Usporadúvanie zlučovaním (merge sort)

- je zrejmé, že jednoprvkové úseky sú vždy usporiadane
- teraz použijeme druhú časť algoritmu: zlúčenie dvoch usporiadaných častí tak, aby aj novovzniknutá časť bola usporiadana, t.j. dostávame časť s dvoma položkami
- podobne sa rozdelí a zlúčí 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 usporiadana celé pole

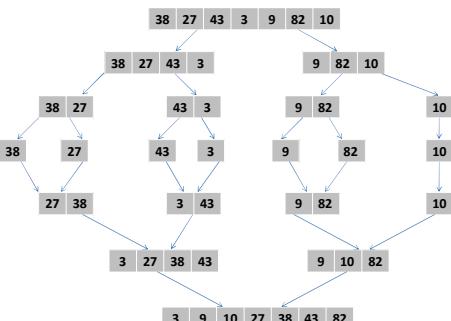
## Usporadúvanie zlučovaním (merge sort)

- 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 načítať tieto údaje do pamäte
  - umožňuje „on-line“ pridávanie ďalších podpostupností (ktoré sa najprv usporiadajú) počas procesu zlučovania.

## Usporadúvanie zlučovaním (merge sort)

- algoritmus môžeme opísť takto:
  - pole 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
  - znova vezmeme prvú z nich a rozdelíme ju na dve, atď. ... až kým nedostaneme jednoprvkové úseky

## Usporadúvanie zlučovaním (merge sort)



## 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. rekurzívne usporiadaj každú podpostupnosť rekurzívnym aplikovaním merge sort-u
  3. zlúč dve usporiadane podpostupnosti do jednej usporiadanej postupnosti.

## zlučovanie

### MERGE()

Assume  $A[p..q]$  and  $A[q+1..r]$  are two sorted.  
Merge( $A, p, q, r$ ) forms a single sorted array  $A[p..r]$ .

**Merge** ( $A, p, q, r$ )  
 $s \leftarrow q - p + 1; t \leftarrow r - q;$   
 $L \leftarrow A[p..q]; R \leftarrow A[q+1..r]$   
 $I[s+1] \leftarrow \infty; R[t+1] \leftarrow \infty$   
 $A[p..r] \leftarrow \text{MergeArray}(L, R)$

### MergeArray

Assume  $L[1:s]$  and  $R[1:t]$  are two sorted arrays of elements: Merge-Array( $L, R$ ) forms a single sorted array  $A[1:s+t]$  of all elements in  $L$  and  $R$ .

**A = MergeArray( $L, R$ )**  
 $\quad I[s+1] \leftarrow \infty; R[t+1] \leftarrow \infty$   
 $\quad i \leftarrow 1; j \leftarrow 1$   
 $\quad \text{for } k \leftarrow 1 \text{ to } s+t$   
 $\quad \quad \cdot \text{do if } I[i] \leq R[j]$   
 $\quad \quad \quad \cdot \text{then } A[k] \leftarrow I[i]; i \leftarrow i+1$   
 $\quad \quad \quad \cdot \text{else } A[k] \leftarrow R[j]; j \leftarrow j+1$

## Správnosť procedúry MergeASrray

### Correctness of MergeArray

- Loop-invariant
  - At the start of each iteration of the **for** loop, the subarray  $A[1:k]$  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  $A$ .

### 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  $A[1:k]$  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  $A$ .

### Inductive Proof of Correctness

- **Maintenance:** (the invariant is true after each iteration)  
WLOG: assume  $L[i] \leq R[j]$ , 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.

### 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.

## Časová výpočtová zložitosť MergeArray

### Complexity of MergeArray

- At each iteration, we perform 1 comparison, 1 assignment (copy one element to  $A$ ) and 2 increments (to  $k$  and  $i$  or  $j$ )
- So number of operations per iteration is 4.
- Thus, Merge-Array takes at most  $4(s+t)$  time.
- **Linear in the size of the input.**

## Časová výpočtová zložitosť usporadúvania zlúčovaním

### Analysis of Merge Sort

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

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

## 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.

## Recursion Tree

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

## Recursion Tree

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

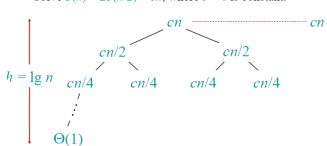
## Recursion Tree

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

$\Theta(1)$

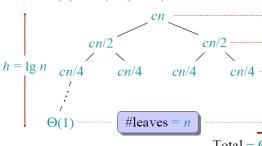
## Recursion Tree

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



## Recursion Tree

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



## 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!

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

31

## Merge Sort (1)

```

type TZoznam = class
 h: Integer;
 next: TZoznam;
end;

procedure rozdel(p: TZoznam; var dp: TZoznam);
var stred, pom: TZoznam;
begin
 if p = nil then // nebolo dô delit
 dp := nil;
 else
 begin
 stred := p;
 pom := stred.next;
 while pom <> nil do // pokial nie je na konci zoznamu
 begin
 pom := pom.next; // posúv sa o jeden krok
 if pom <> nil then // väčšie nie sú na konci
 begin
 stred := stred.next; // posúv stred o jeden krok
 pom := pom.next; // posúv sa o druhý krok
 end;
 end;
 dp := stred.next; // vráti zoznam na stredový prvok
 stred.next := nil; // značkuj dlhší zoznam na polovičný
 end;
 end;
end;

```

## Merge Sort (2)

```

procedure zluc(p1, p2: TZoznam; var p: TZoznam);
var k: TZoznam;
begin
 if p1 = nil then // ak je zoznam p1 prázdny
 p := p2
 else if p2 = nil then // ak je zoznam p2 prázdny
 p := p1
 else // oba zoznamy sú neprázdne
 begin
 if p1.h <= p2.h then // vyriešme prvý prvok nového zoznamu
 begin
 p := p1;
 p1 := p1.next;
 end
 else
 begin
 p := p2;
 p2 := p2.next;
 end;
 end;

```

## Merge Sort (3)

```

// vytváram zoznam p pridávaním na koniec pomocou smerníka k
k := p;
while (p1 <> nil) and (p2 <> nil) do
 if p1.h <= p2.h then
 begin
 k.next := p1;
 k := p1;
 p1 := p1.next;
 end
 else
 begin
 k.next := p2;
 k := p2;
 p2 := p2.next;
 end;
 if p1 = nil then // vyprázdnil sa zoznam p1
 k.next := p2
 else // vyprázdnil sa zoznam p2
 k.next := p1;
end;

```

## Merge Sort (4)

```

procedure MergeSort(var z: TZoznam);
var druhapol: TZoznam;
begin
 if (z <> nil) and (z.next <> nil)
 then // sú aspoň 2 prvky
 begin
 rozdel(z, druhapol);
 MergeSort(z);
 MergeSort(druhapol);
 zluc(z, druhapol, z);
 end;
end;

```

## Usporadúvanie zlučovaním (merge sort)

- rekurzívna procedúra MergeSort spracuje celý zoznam tak, že ho rozdelí na dve časti s približne rovnakým počtom prvkov, na tieto zavolá tú istú procedúru (pokiaľ sú zoznamy viac ako jednoprvkové) a potom takto vzniknuté zoznamy zlúči
- ak máme dobre rozdelený zoznam na dve časti a vraciame sa z rekurzie ostáva nám iba dobre tieto dve časti zlúčiť a to tak, aby aj nový, dlhší zoznam, ostal usporiadany; procedúra zluc teda prechádza oboma zoznamami a vytvára s nich nový

## Usporadúvanie zlučovaním (merge sort)

- procedúra *rozdel* sa bude snažiť nájsť "polovicu" celého zoznamu, ktorý dostáva cez smerník  $p$  a vráti dva zoznamy s približne rovnakým počtom prvkov
- stred sa bude hľadať tak, že po zozname budú "putovať" dva smerníky, jeden po jednom kroku a druhý po dvoch, a ak sa druhý smerník prepracuje na koniec zoznamu, zrejme prvý, ktorý šiel po kroku, bude práve v polovici zoznamu

## Usporadúvanie zlučovaním - zložitosť

- Merge sort využíva dve zásady zlepšenia usporadúvania:
  - menšia postupnosť zaberie menej krokov na usporiadanie než väčšia postupnosť
  - na vytvorenie usporiadanej postupnosti z dvoch už usporiadaných podpostupností, je potrebných menej krokov, než z dvoch neusporiadaných podpostupností.

## Usporadúvanie zlučovaním - zložitosť

- Rôzne verzie usporiadania zlučovaním sa môžu lísiť najmä z hľadiska použitej stratégie zlučovania usporiadaných podpostupností
- No aj ten najznámejší a najjednoduchší spôsob zlučovania zabezpečí najhoršiu časovú zložitosť  $O(n \log n)$  – na rozdiel od quick sort, kde je najhoršia zložitosť  $O(n^2)$

### Running time of MergeSort

Again the running time can be expressed as a recurrence

$$T(n) = \begin{cases} \text{solving_trivial_problem} & \text{if } n=1 \\ \text{num_pieces } T(n/\text{subproblem\_size\_factor}) + \text{dividing} + \text{combining} & \text{if } n>1 \end{cases}$$

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

29

### Repeated Substitution Method

Let's find the running time of merge sort (let's assume that  $n=2^b$ , for some  $b$ ).

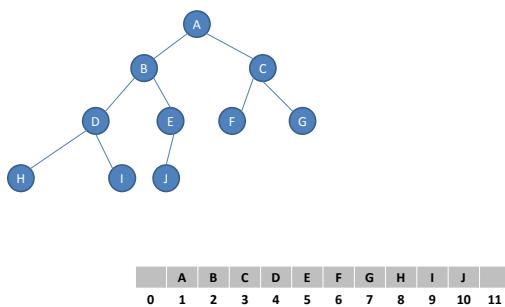
$$\begin{aligned} T(n) &= \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{if } n>1 \end{cases} \\ T(n) &= 2T(n/2) + n \quad \text{substitute} \\ &= 2(2T(n/4) + n/2) + n \quad \text{expand} \\ &= 2^2T(n/4) + 2n \quad \text{substitute} \\ &= 2^2(2T(n/8) + n/4) + 2n \quad \text{expand} \\ &= 2^3T(n/8) + 3n \quad \text{observe the pattern} \\ T(n) &= 2^{\lceil \lg n \rceil}T(n/2^{\lceil \lg n \rceil}) + n \quad \text{Let } 2^{\lceil \lg n \rceil} = m \\ &= 2^{\lceil \lg n \rceil}T(n/m) + n\lg n + n - n\lg n \end{aligned}$$

30

## 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



## Usporadúvanie haldou

- usporadúva prvky pomocou dátovej štruktúry binárna halda
- predstavuje efektívnejšiu verziu usporadúvania výberom
- najväčší (resp. najmenší) prvk 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 prvk poľa) najväčší prvk 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

## Prioritný rad

- každý prvk má prioritu
- prioritný rad - rad zoradený podľa priority
- nie FIFO, ale vyberie sa prvk s najvyššou prioritou
- príklady:
  - súbory na tlač čakajúce v rade
  - procesy čakajúce na preprocesor

## Prioritný rad pomocou spájaného zoznamu

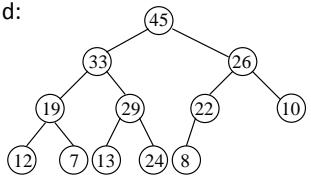
- Pridávanie prvkov na začiatok zoznamu  $O(1)$
- Vymazávanie prvkov - nájdenie prvku s najväčšou prioritou, ten sa vymaže  $O(n)$

## Prioritný rad pomocou BVS

- Pridávanie prvkov - zaradenie do stromu podľa priority (priorita je kľúč)
- Vymazávanie prvkov - vymazanie prvku s najväčšou prioritou, t.j. najpravejší uzol
- Obe operácie -  $O(\log n)$  - výhodnejšie ako pri spájanom zozname
- Nepotrebujeme všetky vlastnosti BVS len na to, aby sme našli prvak s najväčšou prioritou

### Prioritný rad pomocou binárnej haldy

- Binárna halda je binárny strom, pre ktorý platí, že hodnota klúča je väčšia alebo rovná hodnotám klúčov jeho synov
- Príklad:



### Binárna halda

- Binárna halda má menej striktné pravidlá na umiestnenie prvkov ako BVS
- Neplatí, že ľavý podstrom obsahuje prvky s nižšími hodnotami klúčov ako pravý podstrom
- koreň stromu má však vždy najväčšiu hodnotu ( $\geq$  ako ostatné uzly): vymazanie koreňa stromu

### Binárna halda - implemenťacia poľom

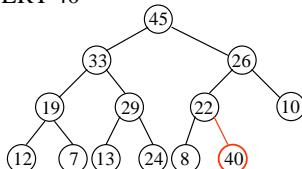
- Koreň stromu na 0-tej pozícii  $heap[0]$
- Deti uzla na i-tej pozícii poľa, ak existujú:
  - $left(i) = 2 * i$
  - $right(i) = 2 * i + 1$
- $heap[i..j]$ , kde  $i \geq 0$ , je binárna halda práve vtedy, keď každý prvok nie je menší ako jeho deti.

### Pridanie prvku do binárnej haldy

- Vytvorí sa nový vrchol na najnižšej úrovni
- Ak hodnota klúča nového uzla  $\leq$  hodnota predchodcu - koniec
- Ak je väčší, vymení sa nový uzol so svojím predchodom
- Ak je hodnota nového uzla väčšia ako nový predchodca, vymení sa aj s ním, ... až pokým nie je strom opäť haldou

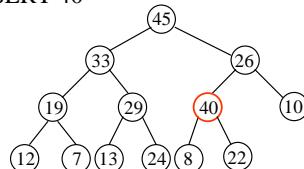
### Pridanie prvku do binárnej haldy - príklad (1)

INSERT 40



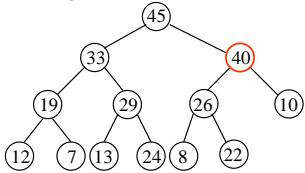
### Pridanie prvku do binárnej haldy - príklad (2)

INSERT 40



### Pridanie prvku do binárnej haldy - príklad (3)

INSERT 40



### Pridanie prvku do binárnej haldy - implementácia

```

INSERT(heap, key)
heap-size (heap) = heap-size(heap) + 1
i = heap-size (heap)
while i > 1 and heap[PARENT(i)] < key
 do heap[i] = heap[PARENT(i)]
 i = PARENT(i)
 heap[i] = key

```

### Priority Queues

```

HEAP-INCREASE-KEY(A, i, key)
1 if $key < A[i]$
2 then error "new key smaller than current"
3 $A[i] \leftarrow key$
4 while $i > 1$ and $A[PARENT(i)] < A[i]$
5 do exchange $A[i] \leftrightarrow A[PARENT(i)]$
6 $i \leftarrow PARENT(i)$

```

### Priority Queues

```

MAX-HEAP-INSERT (A, key)
1 $heap-size[A] \leftarrow heap-size[A] + 1$
2 $A[heap-size[A]] \leftarrow -\infty$
3 HEAP-INCREASE-KEY($A, heap-size[A], key$)

```

### Vymazanie najväčšieho prvku z binárnej haldy (1)

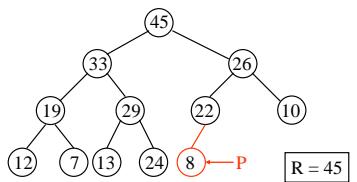
- Odstráni sa koreň haldy, hodnotu kľúča koreňa označíme R
- Odstráni sa najpravejší uzol na najnižšej úrovni (jeho hodnotu označme P)
- Pokúsime sa vyplniť hodnotu koreňa hodnotou P
- Ak hodnota P  $\geq R$ , P sa zapíše do koreňa
- Inak presunieme potomka koreňa s väčšou hodnotou do koreňa,

### Vymazanie najväčšieho prvku z binárnej haldy (2)

- R= hodnota presunutého uzla
- Vzniká voľné miesto, kam sa opäť pokúšame umiestniť P (ak hodnota P  $\geq R$ )
- Takto pokračujeme až pokým nastane hodnota P  $\geq R$ , kde R je hodnota posledného presunutého uzla, alebo posledný presunutý uzol je list - tam presunieme uzol P

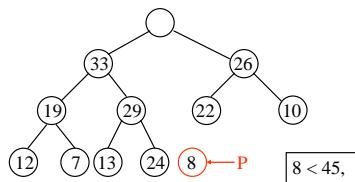
### Vymazanie najväčšieho prvku z binárnej haldy - príklad (1)

EXTRACT-MAX



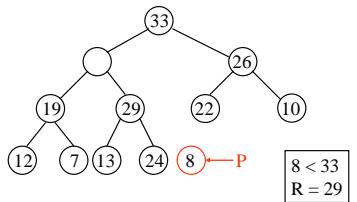
### Vymazanie najväčšieho prvku z binárnej haldy - príklad (2)

EXTRACT-MAX



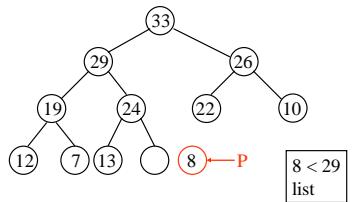
### Vymazanie najväčšieho prvku z binárnej haldy - príklad (3)

EXTRACT-MAX



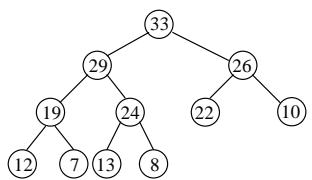
### Vymazanie najväčšieho prvku z binárnej haldy - príklad (4)

EXTRACT-MAX



### Vymazanie najväčšieho prvku z binárnej haldy - príklad (5)

EXTRACT-MAX



### Vymazanie najväčšieho prvku z binárnej haldy - implementácia

EXTRACT-MAX(heap)

if heap-size(heap) < 1

then error

max = heap[0]

heap-size(heap) = heap-size(heap) - 1

HEAPIFY(heap, 0)

return max

## Priority Queues

```

HEAP-EXTRACT-MAX (A)
1 if $heap\text{-size}[A] < 1$
2 then error "heap underflow"
3 $max \leftarrow A[1]$
4 $A[1] \leftarrow A[heap\text{-size}[A]]$
5 $heap\text{-size}[A] \leftarrow heap\text{-size}[A] - 1$
6 MAX-HEAPIFY ($A, 1$)
7 return max

```

Advanced Data Structures, Čaňopek and Ábelová 2005-2007, ITU

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

73

## Udržiavanie haldy HEAPIFY - implementácia (1)

```

HEAPIFY(heap, i)
 $l = left(i)$
 $r = right(i)$
if $l \leq heap\text{-size}(heap)$ and $heap[l] > heap[i]$
 then largest = l
else largest = i
% pokračovanie

```

## Udržiavanie haldy (HEAPIFY) - implementácia (2)

```

if $r \leq heap\text{-size}(heap)$ and $heap[r] > heap[i]$
 then largest = r
if largest $\neq i$
 then exchange $heap[i], heap[largest]$
 HEAPIFY(heap, largest)

```

Advanced Data Structures, Čaňopek and Ábelová 2005-2007, ITU

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

49

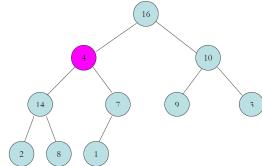
## MAX-HEAPIFY

```

MAX-HEAPIFY (A, i)
 $l \leftarrow LEFT(i)$
 $r \leftarrow RIGHT(i)$
if $l \leq heap\text{-size}[A]$ and $A[l] > A[i]$
 then largest = l
else largest = i
if $r \leq heap\text{-size}[A]$ and $A[r] > A[largest]$
 then largest = r
if largest $\neq i$
 then exchange $A[i]$ with $A[largest]$
 MAX-HEAPIFY ($A, largest$)

```

## MAX-HEAPIFY

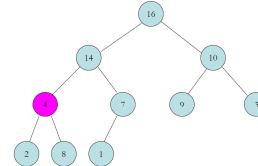


Advanced Data Structures, Čaňopek and Ábelová 2005-2007, ITU

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

50

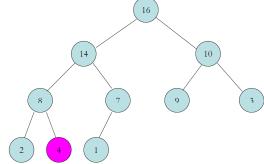
## MAX-HEAPIFY



Advanced Data Structures, Čaňopek and Ábelová 2005-2007, ITU

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

51

**MAX-HEAPIFY**

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

52

**MAX-HEAPIFY, Analysis**

The children's subtrees each have size at most  $2n/3$  – when the last row is exactly  $\frac{1}{2}$  full  
Therefore, the running time is:

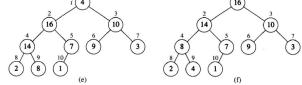
$$T(n) = T(2n/3) + \Theta(1) = O(\lg n)$$

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

53

**Vytvorenie haldy****BUILD-HEAP(heap)**

```
heap-size(heap) = length(heap)
for i = ⌊length[heap] / 2⌋ down to 1
 do HEAPIFY(heap, i)
```



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

56

**Analysis of Building a Heap**

- Since each call to MAX-HEAPIFY costs  $O(\lg n)$  and there are  $O(n)$  calls, this is  $O(n \lg n)$ ...
- Can derive a tighter bound: do all nodes take  $\log n$  time?
- Has at most  $n/2^{h+1}$  nodes at any height (the more the height, the less nodes there are)
- It takes  $O(h)$  time to insert a node of height  $h$ .

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

57

**MAX-HEAPIFY, Analysis**

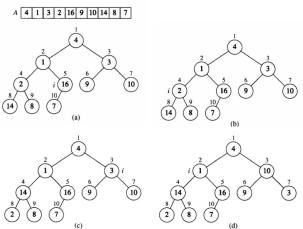
The children's subtrees each have size at most  $2n/3$  – when the last row is exactly  $\frac{1}{2}$  full  
Therefore, the running time is:

$$T(n) = T(2n/3) + \Theta(1) = O(\lg n)$$

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

53

A [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]



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

55

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \leq \sum_{h=0}^{\infty} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = n \sum_{h=0}^{\infty} \frac{h}{2^h} = 2n$$

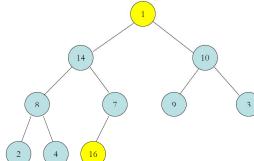
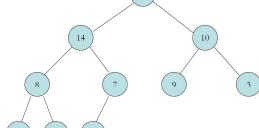
The number of nodes at height h

Multipled by their height

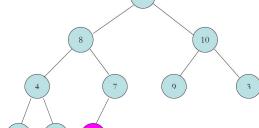
- Thus, the running time is  $2n = O(n)$

## HEAPSORT

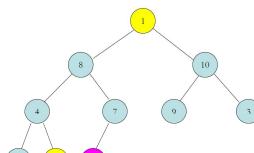
```
HEAPSORT(A)
 BUILD-MAX-HEAP(A)
 for i ← length[A] downto 2
 do exchange A[1] with A[i]
 heap-size[A] ← heap-size[A] - 1
 MAX-HEAPIFY(A, 1)
```



Swap A[1] ↔ A[i]



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



Swap A[1] ↔ A[i]

```

heap-size ← heap-size - 1
MAX-HEAPIFY(4, 1)

```

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

64

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

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

65

```

heap-size ← heap-size - 1
MAX-HEAPIFY(4, 1)

```

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

66

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

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

67

```

heap-size ← heap-size - 1
MAX-HEAPIFY(4, 1)

```

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

68

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

69

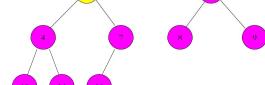
Atd'.

• .....

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

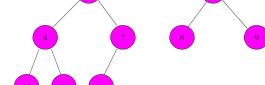
Week 3: Sorting, Insertion, Merge, Heap, Quick

70

heap size  $\leftarrow$  heap size - 1  
MAX\_HEAPIFY(4, 1)

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

71



## Cvičenie

- Implementujte prioritný rad (binárnu haldú) dynamicky.

Pomôcka:

```
typedef struct prioritny_rad {
 int hodnota;
 struct prioritny_rad *lavy, *pravy;
} PRIORITNY_RAD;
```

## 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 haldú 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); // vymen koreň s posledným prvkom
 Dec(posledny); // zmenší rozmer stromu
 uprac_haldu(0, posledny); // oprav strom - koreň je teraz asi zly
 end;
end;
```

### Usporadúvanie haldou

- procedúra `uprac_haldou` vytvára haldou 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 haldou, ale tým, že sme pridali prvok na miesto K, mohla sa haldou K až M pokažť
- preto je potrebné pole znova „vyhaldovať“, t.j. zabezpečiť, aby aj pre K platilo, že má oboch synov menších ako on sám (ak to tak nie je, treba ho zrejmé vymeniť s väčším zo synov a postup opakovat pre tohto syna a príslušný podstrom)
- pomocná procedúra posun posúva prvý parameter na väčšieho syna

### Usporadúvanie haldou - zložitosť

- heap sort má rovnaké časové zložitosť 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 parallelizovať
  - Heap sort nie je stabilný

### Lineárne usporadúvanie

- porovnávacie algoritmy maximálne  $O(n \log n)$
- algoritmy s lineárnom časovou zložitosťou  $O(n)$  používajú iné operácie ako porovnávanie na usporiadanie postupnosti
- použitie distribuovaných algoritmov – algoritmy, kde údaje zo vstupu sú rozdelené do viacerých prechodných štruktúr, ktoré sa potom zhromaždia a umiestnia na výstupe

### Lineárne usporadúvanie

- Výhody a nevýhody oproti porovnávacím algoritmom:
  - lepšia časová zložitosť
  - horšia pamäťová zložitosť – nedá sa pracovať na mieste (in-situ)
  - predpokladá, že vstupné údaje sú z nejakého intervalu

### Usporadúvanie spočítavaním (counting sort)

- určuje počet prvkov menších ako prvok x, pomocou čoho zistí správnu pozíciu prvku x vo vstupnom poli
- predpokladá, že každý prvok z n vstupných prvkov je z nejakého intervalu 1..m.
- ak má byť efektívny, tak musí platiť, že m nie je oveľa väčšie ako n
- vhodný na použitie ak m je malé a kľúče sa často opakujú
- časová zložitosť  $O(n+m)$ 
  - ak m patrí do  $O(n)$ , tak beží v čase  $O(n)$ . Ak m je oveľa väčšie ako n, napr. ak m patrí do  $O(n^2)$ , tak sa berie  $O(m)$ .

### Usporadúvanie spočítavaním

- Pracuje s troma poliami:
  - Pole  $A[]$  obsahuje údaje, ktoré sa majú usporiadať – veľkosť poľa je n
  - Pole  $B[]$  obsahuje konečný usporiadany zoznam údajov – veľkosť poľa je n
  - Pole  $C[]$  je použité na počítanie počtu prvkov – veľkosť poľa je m

## Usporadúvanie spočítavaním

- Fázy algoritmu:

- prvý cyklus inicializuje pole  $C[]$  na nulové hodnoty
- druhý cyklus inkrementuje hodnoty v  $C[]$  podľa početnosti ich výskytu v poli  $A[]$  – početnosť sa zapíše do indexu, ktorý zodpovedá hodnote prvku.
- tretí cyklus pripočíta ku každej hodnote poľa  $C[]$  kumulatívny súčet predchádzajúcich hodnôt tohto poľa
- štvrtý cyklus vypíše usporiadane hodnoty do poľa  $B[]$  – hodnota na indexe i poľa  $C[]$  určuje index v poľu  $B[]$ , do ktorého sa zapíše prvok, ktorý sa rovná indexu  $i$ .

## Usporadúvanie spočítavaním

```
COUNTING-SORT(A, B, k)
1 for i ← 0 to k
2 do C[i] ← 0
3 for j ← 1 to length[A]
4 do C[A[j]] ← C[A[j]] + 1
5 ▷ C[i] now contains the number of elements equal to i.
6 for i ← 1 to k
7 do C[i] ← C[i] + C[i - 1]
8 ▷ C[i] now contains the number of elements less than or equal to i.
9 for j ← length[A] downto 1
10 do B[C[A[j]]] ← A[j]
11 C[A[j]] ← C[A[j]] - 1
```

## Usporadúvanie spočítavaním

| A                                                                                | B                                                                                | C                                                                              |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| 1 2 3 4 5 6 7 8<br>[2 5 3 0 2 3 0 3]                                             | 0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ]                                           | 2 0 2 3 0 1<br>[ ] [ ] [ ] [ ] [ ] [ ]                                         |
| 0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ]                                           | 0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ]                                           | 2 0 2 3 0 1<br>[ ] [ ] [ ] [ ] [ ] [ ]                                         |
| 1 2 3 4 5 6 7 8<br>[2 0 2 3 0 1  ] [ ]                                           | 0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ]                                           | 2 0 2 3 0 1<br>[ ] [ ] [ ] [ ] [ ] [ ]                                         |
| 0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ]                                           | 0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ]                                           | 2 0 2 3 0 1<br>[ ] [ ] [ ] [ ] [ ] [ ]                                         |
| B<br>C<br>(a)                                                                    | B<br>C<br>(b)                                                                    | B<br>C<br>(c)                                                                  |
| 1 2 3 4 5 6 7 8<br>[0 1 2 3 4 5 6 7 8]<br>0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ] | 1 2 3 4 5 6 7 8<br>[0 1 2 3 4 5 6 7 8]<br>0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ] | 1 2 3 4 5 6 7 8<br>[0 0 2 2 3 3 3 5]<br>0 1 2 3 4 5<br>[ ] [ ] [ ] [ ] [ ] [ ] |
| B<br>C<br>(d)                                                                    | B<br>C<br>(e)                                                                    | B<br>C<br>(f)                                                                  |

Figure 8.2 The operation of COUNTING-SORT on an input array  $A[1..8]$ , where each element of  $A$  is a single digit from 0 to 9. (a) The array  $A$  and the auxiliary array  $C$  after line 4. (b) The array  $C$  after line 7. (c)-(f) The output array  $B$  and the auxiliary array  $C$  after one, two, and three iterations of the loop in lines 9-11, respectively. Only the lightly shaded elements of array  $B$  have been filled in. (f) The final sorted output array  $B$ .

Algoritmus zaradujem len kvôli číslam riadkov v analýze zložitosti na dalsom slajde

### Implementation of Counting Sort

```
Algorithm COUNTING-SORT(A, m)
1 n ← A.length
2 initialise array C[1..m]
3 for i ← 1 to n do
4 j ← A[i].key
5 C[j] ← C[j] + 1
6 for i ← 1 to n do
7 C[i] ← C[i] + C[i - 1] ▷ C[i] stores # of keys ≤ i
8 initialise array B[1..n]
9 for i ← n downto 1 do
10 j ← A[i].key ▷ A[i].highest w/ key i
11 B[C[j]] ← A[i] ▷ Insert A[i] into highest
 free index for j keys
12 C[j] ← C[j] - 1
13 for i ← 1 to n do
14 A[i] ← B[i]
```

### Analysis of Counting Sort

- The loops in lines 3-5, 9-12, and 13-14 all require time  $\Theta(n)$ .
- The loop in lines 6-7 requires time  $\Theta(m)$ .
- Thus the overall running time is

$$\Theta(n + m).$$

Note: This does not contradict Theorem 7.3 – that's a result about the **general case**, where keys have an arbitrary size (and need not even be numeric).

\*Note\*: COUNTING-SORT is **STABLE**.  
(After sorting, 2 items with the same key have their initial relative order).

## Radixové usporadúvanie

```
RADIX-SORT(A, d)
```

```
1 for i ← 1 to d
2 do use a stable sort to sort array A on digit i
```

## Koreňové usporadúvanie (Radix Sort)

- Klúč(prvok) sa dá reprezentovať číslom určitého rozsahu
- Radix sort neporovnáva dva klúče, ale spracúvava a porovnáva časti klúčov
- Klúče považuje za čísla zapísané v číselnej sústave so základom M (radix), pracuje s jednotlivými číslicami
- Dokáže usporadúvať čísla, znakové reťazce, dátu, ... (počítače reprezentujú všetký údaje ako postupnosti 1 a 0 – binárna sústava => základ pre radix)

## Koreňové usporadúvanie

- LSD Radix sort (least significant digit) – usporadúvanie cifier postupuje od poslednej cifry (s najmenšou vähou) k prvej číslici (s najväčšou vähou) (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) – stabilný.
- MSD Radix sort – od prvej cifry k poslednej – lexikografické usporiadanie (1, 10, 2, 3, 4, 5, 6, 7, 8, 9) – nestabilný
- Je dôležité na samotné usporadúvanie cifier použiť nejaký stabilný algoritmus, aby sa nemenilo poradie rovnakých usporiadaných cifier jednej vähy pri usporadúvaní cifier inej vähy.
- Kedže počet možných cifier je len 10, tak na ich usporiadanie je výhodné použiť counting sort.

## Koreňové usporadúvanie - príklad

- Usporadúvanie prvkov s trojciferným klúčom:  
(radix sort s M = 10):
  - vytvorí sa 10 kópiek
  - na prvú sa budú dávať prvky s klúčom menším ako 100, na druhú prvky s klúčom z rozsahu 100-199 atď.
  - každú s týchto kópiek následne usporiada rovnakým postupom.

## Koreňové usporadúvanie

|     |     |     |     |
|-----|-----|-----|-----|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 30  | 537 | 830 | 537 |
| 436 | 657 | 535 | 657 |
| 70  | 329 | 537 | 720 |
| 355 | 839 | 657 | 839 |

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

## Koreňové usporadúvanie

### Radix Sort

- Every integer number  $k$  can be represented by at most  $d$  digits in base  $r$  (radix)
  - All digits can be stored in an array  $A[1..d]$
  - $k$  becomes  $A[1]A[2]...A[d]$ , where  $A[i]$  are digits in base  $r$
  - $A[1]$ : the most significant digit
  - $A[d]$ : the least significant digit
- Example
  - Decimal system 015, 155, 008, 319, 325, 100, 111:  $d=3, r=10$
  - For 015:  $A[1]=0, A[2]=1, A[3]=5$
- Idea
  - Sort by the Least Significant Digit first (LSD)
    - the numbers ended with 0 precede the numbers ended with 1, which precede those ended with 2, and so on so forth
  - sort by the next least significant digit
  - continue this until all numbers have been sorted on all  $d$  digits
  - Also can sort by the Most Significant Digit first (MSD)

## Koreňové usporadúvanie

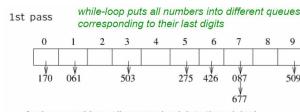
### Radix Sort Algorithm

- RadixSort( $q$ :Queue):Queue
  - for  $i=0$  to  $r-1$  do
    - Make a new queue for the corresponding digit, e.g. 0, 1, 2...
    - $A[i]=\text{Queue make}()$
  - for  $i=r$  downto 1 do / Loop on all digits of each number
    - while ( $\text{NOT } q.\text{isEmpty}()$ ) do
      - $x=q.\text{dequeue}()$
      - $j=i^{\text{th}}$  digit of  $x$ .value from right;
      - $A[j].enqueue(x)$
    - $\text{for } j=0 \text{ to } r-1 \text{ do}$  Put each number into a queue corresponding to its  $i^{\text{th}}$  digit.
      - $q.append(A[j])$
  - $A[i]=\text{Queue.make}()$  Reassemble all numbers back to the original queue. Now they are sorted w.r.t. the  $i^{\text{th}}$  digits.
  - $\text{Intermediate queues are used}$
- We use queues for simplicity of operation
  - All numbers are stored in a queue  $q$
  - Intermediate queues are used

## Koreňové usporadúvanie

### Radix Sort Example (1)

- Least significant digit first
- Numbers to sort: 275, 087, 426, 061, 509, 170, 677, 503



for-loop combines all queues back into the original single queue q, in the order of radices

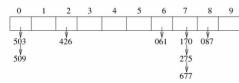
170 061 503 275 426 087 677 509

43

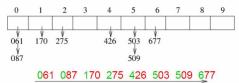
## Koreňové usporadúvanie

### Radix Sort Example (2)

2nd pass 170 061 503 275 426 087 677 509



3rd pass 503 509 426 061 170 275 677 087



in sorted order

44

## Iný príklad

### Radix Sort (cont'd)

**Idea**  
Sort the keys digit by digit, starting with the least significant digit.

**Example**

|     |     |     |     |
|-----|-----|-----|-----|
| now | sob | tag | ace |
| for | nob | bet | bet |
| tip | ace | bet | dim |
| ilk | tag | dim | for |
| dim | ilk | tip | hut |
| tag | dim | sky | ilk |
| ilk | tip | ilk | jot |
| bet | tip | ilk | bet |
| nob | for | sob | nob |
| nob | nob | now | now |
| sky | bet | for | sky |
| hut | bet | jet | sob |
| ace | now | now | tag |
| bet | sky | hut | tip |

AADS Lecture 8

Mary Cogen

### Radix Sort Analysis

- Loop invariant of the main for-loop (`for i:=d downto 1 do`)
  - The queue elements are sorted according to their last  $d-i$  digits
- Increasing the base  $r$  decreases the number of passes
- $d$  becomes smaller
- Running time (input size  $n$ )
  - $d$  passes over the numbers
  - each pass takes  $|q| \cdot r^d$ 
    - while (`NOT q.isEmpty()`) do
      - for  $j=0$  to  $r-1$  do
    - total:  $O(|q| \cdot r^d)$
    - worst case if  $|q|=O(n)$ :  $O((n+r)^d)$ 
      - Or as  $r$  and  $d$  are constants
  - Remarks
    - Radix sort is not based on comparison; the values are used as array indices when locating corresponding queues
    - Radix sort is good for sorting long sequences of small numbers
      - Large  $n$ , fixed (small)  $d$  and  $r$

45

## Koreňové usporadúvanie (1)

```
const n=10;
type pole=array[1..n] of integer;
ppole=record
 p: array[0..n] of integer;
 poc:integer;
end;

procedure RadixSort (var p:ppole);
var pom: array[0..9] of ppole;
 i,j,k,l,p1,moc:integer;
begin
 moc:=1;
```

## Koreňové usporadúvanie (2)

```
for i:=1 to 5 do
begin
 moc:=moc*10;
 for L:=0 to 9 do
 pom[L].poc:=0;
 k:=1;
 for j:=1 to n do
begin
 p1:= p[j] mod moc;
 p1:=p1 div (moc div 10);
 inc(pom[p1].poc);
 pom[p1].p[pom[p1].poc]:=p[j];
end;
```

### Koreňové usporadúvanie(3)

```

for L:=0 to 9 do
 for j:=1 to pom[L].poc do
 begin
 p[k]:=pom[L].p[j];
 inc(k);
 end;
 end;
end;

```

### Koreňové usporadúvanie

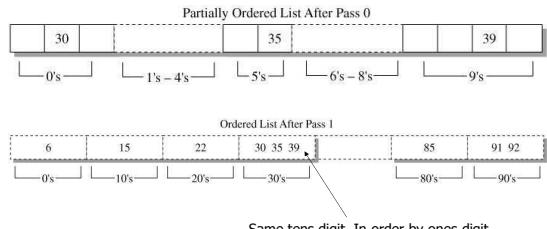
- Lineárny algoritmus, ktorý usporadúva pole čísel. Využíva postupnosť jednotlivých číslic pre čiatočné usporiadanie.
- Elementy v poli sú presúvané do poľa 10 frontov, pričom o zarádení rozhoduje zvolená číslica. Postupným presúvaním sa vytvára čiastočne usporiadany zoznam. Algoritmus končí, keď je vstupné pole usporiadane

### Radix Sort príklad

- vstup: [91, 6, 85, 15, 92, 35, 30, 22, 39]
- 
- po prechode 0: [30, 91, 92, 22, 85, 15, 35, 6, 39]
- 
- po prechode 2: [6, 15, 22, 30, 35, 39, 85, 91, 92]

### Radix Sort (continued)

Initial sequence: {91, 6, 85, 15, 92, 35, 30, 22, 39}



### Koreňové usporadúvanie

- Radix sort používa desiatkovú reprezentáciu čísla:  

$$\text{hodnota} = x_{d-1}10^{d-1} + x_{d-2}10^{d-2} + \dots + x_210^2 + x_110^1 + x_010^0$$
  - Implementácia frontov
- ```

LinkedQueue[] digitQueue = new LinkedQueue[10];
for (i=0;i < digitQueue.length;i++)
    digitQueue[i] = new LinkedQueue();

```

Koreňové usporadúvanie

```

// podporná metóda pre radixSort()
// distribuuje jednotlivé elementy do 10 frontov
// power = 1 ==> 1's digit
// power = 10 ==> 10's digit
// power = 100 ==> 100's digit
// ...

private static void distribute(int[] arr,
    LinkedQueue[] digitQueue, int power)
{
    int i;
    for (i = 0; i < arr.length; i++)
        digitQueue[(arr[i] / power) % 10].push(arr[i]);
}

```

Koreňové usporadúvanie

```
// podporná metóda pre radixSort()
// vyberá prvky z frontov a vkladá ich
// späť do pôvodného pola

private static void collect(
    LinkedQueue[] digitQueue, int[] arr)
{
    int i = 0, digit;

    for (digit = 0; digit < 10; digit++)
        while (!digitQueue[digit].isEmpty())
        {
            arr[i] = digitQueue[digit].pop();
            i++;
        }
}
```

radixSort()

```
public static void radixSort(int[] arr, int d)
{
    int i;
    int power = 1;
    LinkedQueue[] digitQueue = new LinkedQueue[10];
    for (i=0; i < digitQueue.length; i++)
        digitQueue[i] = new LinkedQueue();
    for (i=0; i < d; i++)
    {
        distribute(arr, digitQueue, power);
        collect(digitQueue, arr);
        power *= 10;
    }
}
```

Koreňové usporadúvanie - zložitosť

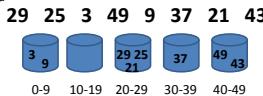
- rieši väčšie časové nároky counting sortu, keď sa pracuje s veľkými číslami
- ak sa pracuje s desiatimi číslami ($n = 10$), ktoré majú 5 cifier ($k = 5$), tak potom m ($m = 10^5 - 1$) značne prevyšuje n a čas. zložitosť counting sort je preto $O(m)$
- pri radix sort sa m rozloží na jednotlivé cifry, čím sa zmenší ich rozsah ($n = 10$, $m = 10$) a čas. zložitosť counting sort cifier je tak $O(n)$ a celková časová zložitosť radix sort je $O(k.n)$

Vedierkové usporadúvanie (Bucket sort)

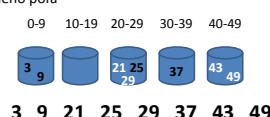
- Predpokladá, že vstup je generovaný náhodným procesom, ktorý prvky distribuuje rovnomerne na celom intervale.
- Rozdelí interval na n rovnako veľkých disjunktných podintervalov (vedierok - bucketov) a potom do nich rozmiestní vstupné čísla
- osobitne v každom vedierku sa potom tieto čísla usporiadajú.

Vedierkové usporadúvanie

- Vytvoria sa prázdne vedierka veľkosti M/n (M – maximálna hodnota vstupného poľa, n – počet prvkov vstupného poľa)
- Rozptylenie – prechádzanie vstupným poľom a rozmiestnenie každého prvku do príslušajúceho vedierka



- Usporiadanie naplnených vedierok
- Zreťaženie vedierok – postupné prechádzanie usporiadaných vedierok a presúvanie prvkov späť do vstupného poľa



Vedierkové usporadúvanie

```
BUCKET-SORT(A)
1   n ← length[A]
2   for i ← 1 to n
3       do insert A[i] into list B[1..nA[i]]
4   for i ← 0 to n - 1
5       do sort list B[i] with insertion sort
6   concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```

Vedierkové usporadúvanie

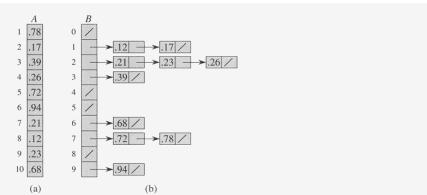


Figure 8.4 The operation of BUCKET-SORT. (a) The input array $A[1 \dots 10]$. (b) The array $B[0 \dots 9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10]$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \dots, B[9]$.

Vedierkové usporadúvanie - zložitosť

- jednotlivé vedierka väčšinou predstavujú spájaný zoznam, do ktorého sa na správne miesto presúvajú prvky zo vstupného poľa (insert sort)
- činnosti ako vytvorenie vedierok, určenie prislúchajúceho vedierka, presunutie prvku do vedierka a zrežazenie vedierok do výslednej postupnosti trvajú $O(n)$
- časové usporiadanie prvkov vo vedierkach insert sortom trvá $O(n^2)$

Vedierkové usporadúvanie - zložitosť

- výsledná časová zložitosť závisí od rozloženia prvkov vo vedierkach. Ak sú prvy rozmiestnené nerovnomerne a v niektorých vedierkach ich je veľmi veľa, tak časová zložitosť insert sortu $O(n^2)$ prevažuje nad lineárnu zložitosťou a predstavuje výslednú zložitosť celého usporadúvania
- takýto stav sa môže vyskytnúť ak rozsah prvkov m je oveľa väčší ako ich počet
- preto sa niekedy celková zložitosť značí podobne ako pri counting sorte $O(n+m)$. Ak $m=O(n)$, tak výsledná časová zložitosť je $O(n)$
- ak sa počet vedierok rovná počtu vstupných prvkov, tak v priemere to vychádza na jeden prvok v každom vedierku, a preto sa za priemernú zložitosť berie $O(n)$

Porovnanie jednotlivých metód

Podľa časovej zložitosti

	priem.	najhoršia
Vkladaním	$O(N^2)$	$O(N^2)$
Výmenou	$O(N^2)$	$O(N^2)$
Výberom	$O(N^2)$	$O(N^2)$
Shellovo	$O(N \log^2 N)$	$O(N^2)$
QuickSort	$O(N \log N)$	$O(N^2)$
MergeSort	$O(N \log N)$	$O(N \log N)$
HeapSort	$O(N \log N)$	$O(N \log N)$
CountingSort	$O(N + M)$	$O(N + M)$
RadixSort	$O(k.N)$	$O(k.N)$
BucketSort	$O(N)$	$O(N^2)$

Podľa časovej zložitosti a stability

	pam. zložitosť	stabilný
Vkladaním	$O(1)$	áno
Výmenou	$O(1)$	áno
Výberom	$O(1)$	áno
Shellovo	$O(1)$	nie
QuickSort	$O(\log N)$	nie
MergeSort	$O(N)$	áno
HeapSort	$O(1)$	nie
CountingSort	$O(N + M)$	áno
RadixSort	$O(N)$	áno(LSD)
BucketSort	$O(N)$	áno

Porovnanie jednotlivých metód

- aj napriek tomu, že distribuované algoritmy usporadúvania majú v priemere lineárnu zložitosť, tak kvôli vyššej rézii niektorých krokov (extrahovanie cifier, kopírovanie polí) sú v praxi väčšinou pomalšie ako porovnávacie algoritmy usporadúvania s priemernou časovou zložitosťou $O(n \log n)$