

10. přednáška

Řazení I.

Rozpis látky

- Klasifikace principů řazení.
- Řazení na principu výběru - Bubble-sort a jeho varianty, Heap sort.
- Řazení polí na principu vkládání, "Bubble-Insert sort", "Binary-insert sort".
- "Quick sort";
- "Shell sort"; "Merge sort"; "List merge sort", "Radix sort".

Klasifikace algoritmů řazení

- Podle přístupu k paměti:
 - Metody vnitřního řazení (metody **řazení polí**).
Přímý (náhodný) přístup.
 - Metody vnějšího řazení (**řazení souborů**).
Sekvenční přístup. Řazení seznamů.
- Podle typu procesoru:
 - **sériové** (jeden procesor) – jedna operace v daném okamžiku
 - **paralelní** (více procesorů) – více souběžných operací

- Podle principu řazení:
 - Princip **výběru** (selection) - přesouvají maximum/minimum do výstupní posloupnosti
 - Princip **vkládání** (insertion) – vkládají postupně prvky do seřazené výst. posloupnosti
 - Princip **rozdělování** (partition) – rozdělují postupně množinu prvků na dvě podmnožiny tak, že prvky jedné jsou menší než prvky druhé
 - Princip **slučování** (merging) setřídí postupně seřazené dvě podmnožiny do jedné
 - Jiné principy ...

Smluvené konvence

- V následujících partiích budou metody řazení v polích vykládány na silně zjednodušené struktuře množiny dat, implementované polem s jednosložkovými položkami, představovanými klíčem typu integer:

type

TA=array[1..N] of integer;

var

A:TA;

Toto pole bude vstup/výstupním (varovým) parametrem procedury řazení nebo jejím globálním objektem.

Řazení na principu výběru (Select sort)

Řazení na principu výběru je nejjednodušší a asi nepřirozenější způsob řazení. Jejím jádrem je cyklus pro vyhledání pozice extrémního (minimálního/maximálního) prvku v zadaném segmentu pole:

...

Princip lze popsat takto:

```
i := 1 ;
```

```
for i := i to N do begin
```

```
    (* najdi minimální prvek pole mezi indexy i a n. Polohu  
       (index) minima ulož do pomocné proměnné PInd *)
```

```
    A[i] := A[PInd] ;
```

```
end
```

```

procedure SelectSort(var A:TA);
var i,j,PInd, PMin:integer;
begin
    for i:=1 to N-1 do begin
        PInd:=i;    (* Poloha pomocného
minima*)
        PMin:=A[i];  (* Pomocné minimum*)
        for j:=i+1 to N do
            if PMin>A[j]
            then begin
                PMin:=A[j];
                PInd:=j
            end;
            A[i] :=: A[PInd]
        end (* for *)
    end;

```

12.9.2014

- Metoda je nestabilní. (Vyměněný první prvek se může dostat „za“ prvek se shodnou hodnotou).
- Má kvadratickou časovou složitost
- Experimentálně byly naměřeny výsledky:
(kde OSP je opačně seřazené pole a NUP je náhodně uspořádané pole, N je počet prvků.)

N	128	256	512
OSP	64	254	968
NUP	50	212	774

Metoda „bublinového výběru“ – „Bubble-sort“

- Princip je shodný, ale metoda nalezení extrému je jiná. Porovnává se každá dvojice a v případě obráceného uspořádání se přehodí. Při pohybu zleva doprava se tak maximum dostane na poslední pozici. Minimum se posune o jedno místo směrem ke své konečné pozici.

Veselé příklady na webu

bubble sort

- http://www.youtube.com/watch?v=t_xkgcakREw

merge sort

- http://www.youtube.com/watch?v=O50M24pSmTs&feature=PlayList&p=BF8233522D3557E2&playnext=1&playnext_from=PL&index=5

```

procedure BubbleSort(var A:TA);
var
    i,j: integer;
    Konec:Boolean;
begin
    i:=2;
    repeat
        Konec:=true;
        for j:=N downto i do (* Bublínový cyklus*)
            if A[j-1] > A[j]
            then begin
                A[j-1] := A[j];
                Konec:=false
            end;
            i:=i+1
        until Konec or (i=(N+1));
    end;

```

```

procedure BubbleSelect (var A:TA); (* Jiná varianta *)
var
    i, PomN:integer;
    Pokracuj:Boolean;
begin
    PomN:=N; Pokracuj:=true;
    while Pokracuj and (PomN>1) do begin
        Pokracuj:=false;
        for i:=1 to PomN -1 do (* bublinový cyklus*)
            if A[i+1] < A[i]
                then begin
                    A[i+1] :=: A[i];
                    Pokracuj:=true (* výměna – nelze skončit *)
                end; (* if *)
        PomN:=PomN-1;
    end;
end;

```

- Bublínový výběr je metoda **stabilní** a přirozená. Je to jedna z mála metod použitelná pro vícenásobné řazení podle více klíčů!!!
- Má časovou složitost kvadratickou.
- Je to nejpobulárnější a nejméně efektivní metoda. Je to nejrychlejší metoda v případě, že pole je již seřazené!
- Experimentálně naměřené hodnoty

n	256	512
NUP	338	1562
OSP	558	2224

Od Bubble-sortu byla odvozena řada vylepšovacích variant, bez valného účinku. Jsou ale programátorsky zajímavé.

- Ripple-sort si pamatuje polohu první výměny a je-li větší než 1 neprochází dvojicemi u nichž je jasné, že se nebudou vyměňovat.
- Shaker-sort střídá směr probublávání zleva a zprava (používá „houpačkovou metodu“) a skončí uprostřed.
- Shuttle-sort „zavede“ při výměně dvojice menší prvek zpět „na své“ místo a pak pokračuje dál. Končí tím, že nevymění nejpravější dvojici.

Řazení hromadou - Heap Sort

- Hromada je struktura stromového typu, pro niž platí, že mezi otcovským uzlem a všemi jeho synovskými uzly platí stejná relace uspořádání (např. otec je větší než všichni synové). Nejčastější případ hromady – heapu je binární heap, založený na binárním stromu.

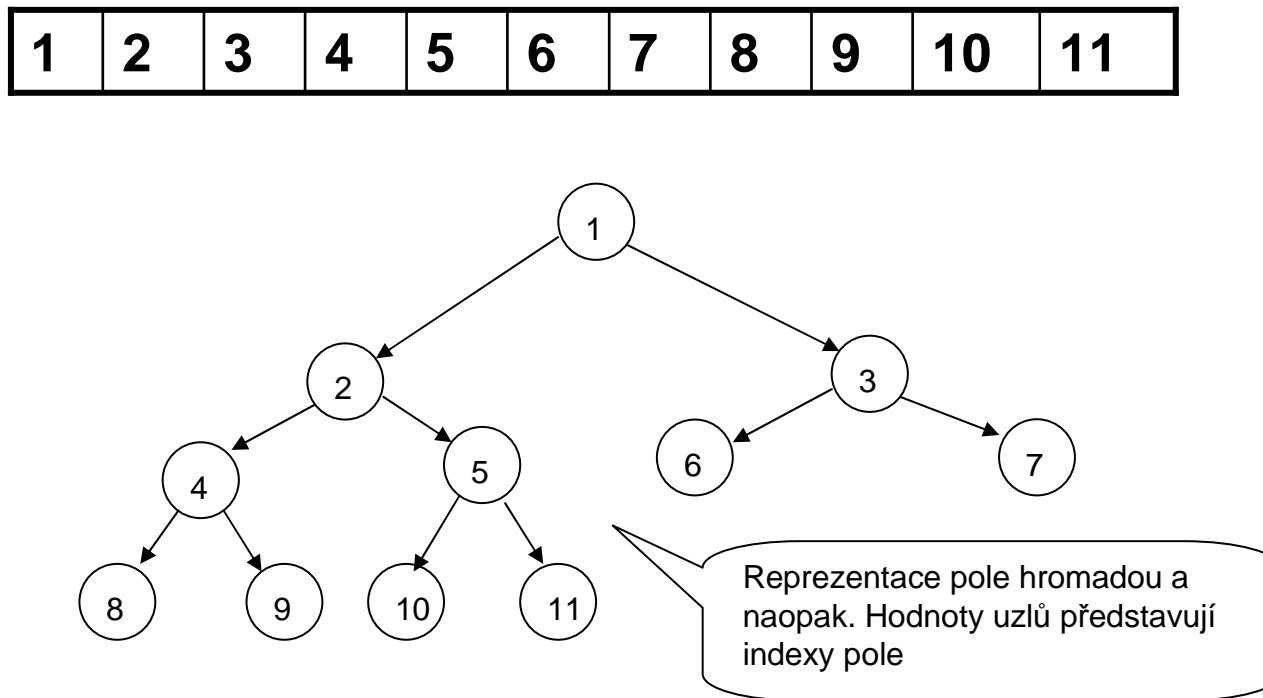
Rekonstrukce hromady

- Významnou operací nad hromadou je její rekonstrukce poté, co se poruší pravidlo hromady v jednom uzlu.
- Nejvýznamnějším případem je porušení v kořeni. Operaci, která znovuustaví hromadu porušenou v kořeni říkáme „Sift“ (prosetí), nebo také „zatřesení hromadou“. Spočívá v tom, že prvek z kořene postupnými výměnami propadne na „své“ místo a do kořene se dostane prvek, splňující pravidla hromady. Tato operace má v nejhorším případě složitost $\log_2 n$.

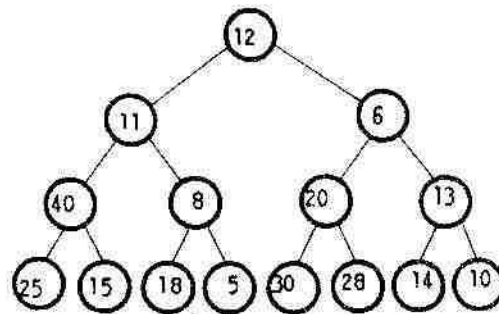
- Jinými slovy, kdyby hromada měla v kořeni nejmenší ze všech prvků pak kdybych odebíral z kořene prvky a vkládal je do výstupního pole, a po každém odebrání do kořene vložil hodnotu nejnižšího a nejlevějšího uzlu a poté hromadou „zatřásl“, získal bych seřazenou posloupnost s lineární složitostí $n \cdot \log_2 n$. To je princip Heapsortu.

Binární strom o n úrovních (a také hromadu), který má všechny uzly na všech $(n-1)$ úrovních a na nejvzdálenější n -té úrovni má všechny uzly zleva, lze snadno implementovat polem. Pak platí pro otcovský a synovské uzly vztah:

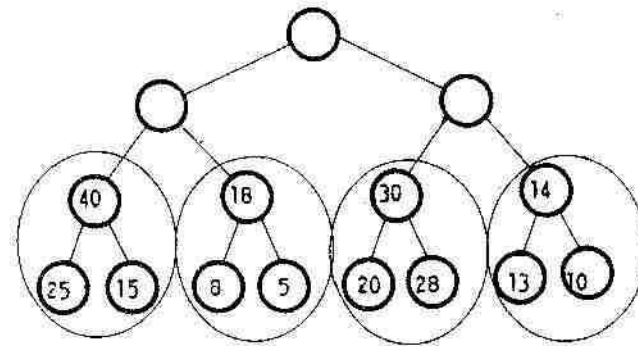
když je otcovský uzel na indexu i , pak je levý syn na indexu $2i$ a pravý syn na indexu $2i+1$.



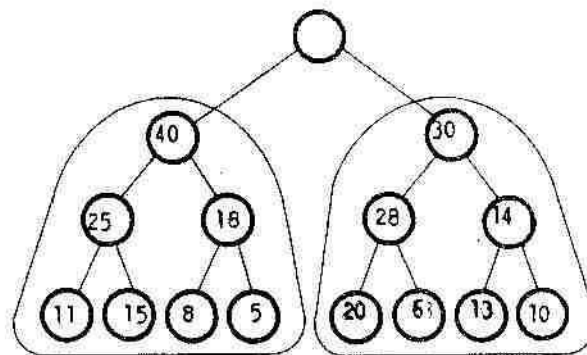
Mějme proceduru SiftDown, která znovuustaví hromadu porušenou v kořeni - proseje prvek v kořeni, který jako jediný porušuje pravidlo hromady.



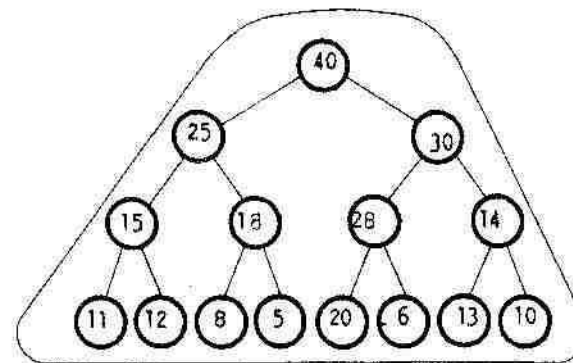
Neuspořádané pole



Procedura sift vytvořila 4 hromady (zprava doleva) na předposlední (3.) úrovni



Procedura sift vytvořila 2 hromady (zprava doleva) na 2. úrovni



V posledním kroku vytvoří procedura ze dvou hromad jedinou hromadu

Pak hromadu lze ustavit tak, že napřed ustavím „porušenou“ hromadu, jejíž kořen je nejpravější a nejnižší „otcovský uzel“ a postupuji ustavováním po všech uzlech doleva a nahoru až k hlavnímu kořeni, jak to ilustroval předcházející obrázek. Má-li pole N prvků, pak nejnižší a nejpravější uzel odpovídající hromady má index $N \text{ div } 2$.

(* Ustavení hromady *)

```
Left := N div 2; (* index nejnižšího a nejpravějšího  
uzlu *)
```

```
Right := N;
```

```
for i := Left downto 1 do  
  SiftDown(A, i, Right);
```

Celý algoritmus Heapsortu má pak tvar:

```
procedure HeapSort(var A:TA);
var
    i, Left, Right: integer;
begin (* Ustavení hromady *)
    Left := N div 2; (* ind. nejnižšího a nejpravějšího uzlu *)
    Right := N;
    for i := Left downto 1 do SiftDown(A, i, Right);
    (* Vlastní cyklus Heap-sortu *)
    for Right := N downto 2 do begin
        A[1] := A[Right]; (* Výměna kořene s akt. posledním
prvkem *)
        SiftDown(A, 1, Right - 1)    (* Znovuustavení hromady
    *)
    end; (* for *)
end; (* procedure *)
```

Při implementaci binárního stromu polem je důležité poznat konec větve (terminální uzel, nebo uzel který nemá pravého syna):

- Je-li dvojnásobek indexu zvýšený o 1 roven počet prvků pole N, pak odpovídající uzel má pravého syna. V opačném případě ho nemá.
- Je-li dvojnásobek indexu (menší nebo) roven počtu prvků, pak odpovídající uzel nemá pravého syna ale má pouze levého syna.
- Je-li dvojnásobek uzlu větší než N, pak odpovídající uzel je terminální.

```
procedure SiftDown(var A:TArr;Left,Right:integer);  
(* Left je kořenový uzel s porušující heap, Right je velikost pole *)  
var  
    i,j:integer;  
    Cont:Boolean; (* Řídící proměnná cyklu *)  
    Temp:integer; (* Pomocná proměnná téhož typu jako  
položka pole *)  
begin  
    i:=Left;  
    j:=2*i; (* Index levého syna *)  
    Temp:=A[i];  
    Cont:=j<=Right; (* pokračování na další straně *)
```

```

while Cont do begin
    if j<Right
    then (* Uzel má oba synovské uzly *)
    if A[j]< A[j+1]
    then (* Pravý syn je větší *)
        j:=j+1; (* nastav jako většího z dvojice synů *)
    if Temp >= A[j]
    then (* Prvek Temp již byl posunut na své místo; cyklus končí *)
        Cont:=false
    else begin (* Prvek Temp propadá níž, A[j] jde o úroveň výš *)
        A[i]:=A[j]; (* *)
        i:=j;      (* syn se stane otcem pro příští cyklus *)
        j:=2*i;    (* příští levý syn *)
        Cont:=j<=Right; (* podmínka : "j není terminální uzel *)
    end (* if *)
end; (* while *)
A[i]:=Temp; (* final positioning of the sifted root *)
end; (* procedure *)

```

- Heapsort je řadící metoda s lineární složitostí, protože „sift“ umí rekonstruovat hromadu (najít extrém mezi N prvky) s logaritmickou složitostí. Tato vlastnost může být významná i pro jiné případy.
- Heapsort je nestabilní a nechová se přirozeně.
- Naměřené hodnoty:

N	256	1024
SP	42	210
NUP	38	186
OSP	40	196

K domácímu procvičení

Je dána hromada o N prvcích typu integer v poli H . Napište proceduru, která rekonstruuje (znovuustaví) hromadu porušenou zápisem libovolné hodnoty na index $1 \leq K \leq N$.

Řazení na principu vkládání

- Řazení vkládáním (insert) se podobá mechanismu, kterým hráč karet bere postupně karty ze stolu a vkládá je do uspořádaného vějíře v ruce.
- Necht' je pole rozděleno na dvě části: levou – seřazenou a pravou neseřazenou. Levou část tvoří na začátku první prvek. Pak má řazení strukturu:

for i:=2 to N do begin

(* najdi v levé části index K, na který se má zařadit prvek A[i]*);

(* posuň část pole od K do i-1 o jednu pozici doprava *);

(* vlož na A[k] hodnotu zařazovaného prvku *)

end

Metoda bublinového vkládání – Bubble-insert sort

- Bubble insert kombinuje vyhledání místa pro vkládání i posun segmentu pole v jednom cyklu postupným porovnáváním a výměnou dvojic prvků. Tím se podobá stejnojmenné metodě výběru.

```

procedure BubbleInsertSort(var A:TArray) ;
var
    i,j:integer;
    Tmp:integer;
begin
    for i:=2 to N do begin
        Tmp:=A[i];
        A[0]:=Tmp; (* Ustavení zářky *)
        j:=i-1;
        while Tmp<A[j] do begin (* najdi a posuň prvek *)
            A[j+1]:=A[j];
            j:=j-1
        end; (* while *)
        A[j+1]:=Tmp; (* konečné vložení na místo *)
    end (* for *)
end; (* procedure *)

```

Analýza metody

- Metoda je stabilní, chová se přirozeně a pracuje in situ. (Je vhodná pro vícenásobné řazení podle více klíčů).
- Má kvadratickou časovou složitost.
- Experimentálně byly naměřeny tyto hodnoty:

n	256	512	1024
SP	4	6	14
NUP	156	614	2330
OSP	312	1262	5008

Vkládání s binárním vyhledáváním

- Protože se vyhledává místo pro vkládání v seřazeném poli, lze snadno dospět k námětu vyhledávat nikoli sekvenčně, ale binárně. Vyhledávací metoda musí v případě shodných klíčů nalézt místo za nejpravějším ze shodných klíčů.

```

procedure BinaryInsertSort(var A:TArr);
var
    i,j,m,Left,Right,Insert:integer;
    Tmp:integer;
begin
    for i:=2 to N do begin
        Tmp:=A[i];
        Left:=1; Right:=i-1; (* nastavení levého a pravého ind. *)
        while Left <=Right do begin (* stand. bin. vyhled. *)
            m:=(Left+Right)div 2;
            if Tmp<A[m]
            then Right:=m-1
            else Left:= m+1;
        end;
        for j:=i-1 downto Left do
            A[j+1]:=A[j]; (* posun segmentu pole doprava *)
        A[Left]:=Tmp;
    end; (* for *)
end; (* procedure *)

```

Analýza

- Metoda je stabilní, chová se přirozeně a pracuje in situ.
- Má kvadratickou časovou složitost.
- Binární vyhledávání vkládací princip výrazně nevylepšilo :

N	256	512	1024
NUP	134	502	1024
OSP	248	956	3736

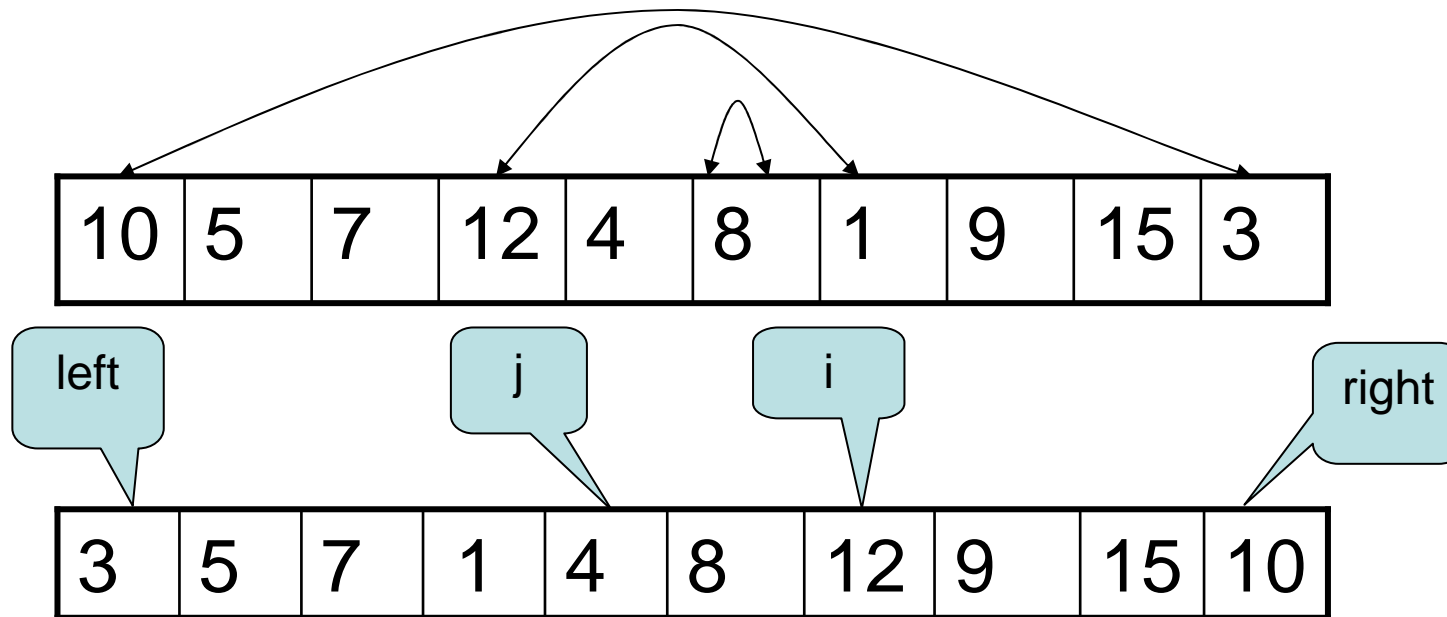
Řazení rozdělováním – Quick Sort

- Princip: Kdyby existoval algoritmus, který umí (rychle) rozdělit množinu položek na dvě podmnožiny, z nichž jedna by obsahovala všechny prvky s klíčem menším (nebo rovným) jisté hodnotě a druhá by obsahovala všechny prvky s klíčem větším (nebo rovným) téže hodnotě, pak se tento algoritmus mohl postupně aplikovat na každou z podmnožin tak dlouho, dokud by obsahovala více než jeden prvek. Výsledkem by byla seřazená množina. Mechanismu rozdělení se také říká „partition“.

Mechanismus rozdělení – „partition“

`procedure partition (left, right:integer; var i,j:integer);`

Procedura rozdělí následující pole na dvě části, left..j a i..right, které obsahují hodnoty menší resp. větší než 8.



Rekurzivní zápis mechanismu QuickSort má tvar:

```
procedure QuickSort(var A:TArr; left, right:integer);  
(* Při volání má left hodnotu 1 a right hodnotu N *)  
var i,j:integer;  
begin  
    Partition(A,left,right,i,j);  
    if left<j then  
        QuickSort(A, left, j); (* Rekurze doleva *)  
    if i<right then  
        QuickSort(A, i, right); (* Rekurze doprava *)  
end;
```

Tajemství vysoké rychlosti Quicksortu je skryto v mechanismu „**partition**“. Jeho autorem je C.A.R.Hoare, významná osobnost v oboru teorie a tvorby programů.

Medián daného souboru čísel je hodnota, pro níž platí, že polovina čísel v souboru je větší a polovina je menší. Kdybychom znali hodnotu mediánu, mohli bychom použít tento mechanismus:

Procházím pole zleva a najdu první číslo, které je větší než medián. Pak procházím zprava a najdu první číslo, které je menší než medián. Tato dvě čísla mezi sebou vyměním a pokračuji v hledání dalšího většího čísla zleva a dalšího menšího zprava. Proces ukončím, až se dva indexy (i jdoucí zleva a j jdoucí zprava) překříží. Tím algoritmus „**partition**“ končí a indexy i a j jsou výstupními parametry vymezujícími intervaly $left..j$ a $i..right$.

Hoare vtisknul algoritmu „**partition**“ dvě významné vlastnosti:

- 1) Protože stanovení mediánu je náročné, nahradil hodnotu mediánu „libovolnou“ hodnotou z daného souboru čísel. Pracujeme s ní jako s mediánem a říkáme jí „pseudomedián“. Nejvhodnější pro tuto roli je číslo ze středu intervalu - $(\text{left} + \text{right}) \div 2$. Experimentálně je prokázáno, že toto číslo splní svou roli velmi podobně jako medián.
- 2) Hoare použil pseudomedián jako „zarážku“ a tím si ušetřil kontrolu konce pole. (Co by se stalo, kdyby pole bylo naplněno stejnými čísly – pseudomedián my byl také toto číslo a při hledání prvního většího zleva by algoritmus musel kontrolovat pravý okraj pole...). Hoare hledal první hodnotu zleva, která je větší **nebo rovna**. A následně zprava, která je menší **nebo rovna**. Rovnost způsobí, že pseudomedián funguje jako zarážka:

```

procedure partition(var A:TArr; left,right:integer; var
i,j:integer);
var
    PM:integer; (* pseudomedián *)
begin
    i:=left;  (* inicializace i *)
    j:=right; (* inicializace j *)
    PM:=A[(i+j) div 2]; (* ustavení pseudomediánu *)
    repeat
        while A[i] < PM do i:=i+1; (* první i zleva, pro A[i]>=PM *)
        while A[j] > PM do j:=j-1; (* první j zprava pro A[j]<=PM *)
        if i<=j
            then begin
                A[i]:=A[j]; (* výměna nalezených prvků *)
                i:=i+1;
                j:=j-1
            end
        until i>j; (* cyklus končí, když se indexy i a j překříží *)
    end; (* procedure *)

```

Nerekurzivní zápis Quicksortu

Nerekurzivní zápis Quicksortu využívá zásobník. Mechanismus rozdělení partition rozdělí dané pole na dva segmenty. Jeden z nich se podrobí dalšímu dělení a hraniční indexy druhého se uchovají v zásobníku.

Algoritmus sestává ze dvou cyklů. Vnitřní cyklus provádí opakované dělení segmentu pole a uchovávání hraničních bodů druhého segmentu v zásobníku. Jakmile „není co dělit“, cyklus se ukončí a opakuje se vnější cyklus. Vnější cyklus vyzvedne ze zásobníku hraniční body dalšího segmentu a vstoupí do vnitřního cyklu. Vnější cyklus se ukončí, když je zásobník prázdný a není žádný další segment k dělení.

```

procedure NonRecQuicksort (left,right:integer);
var
    i,j:integer;
    S:TStack; (* deklarace ADT zásobník *)
begin
    SInit(S); (* inicializace zásobníku *)
    Push(S,left); (* vložení levého ind, prvního segmentu *)
    Push(S,right); (* vložení pravého ind. prvního segmentu *)
    while not Sempty do begin (* vnější cyklus *)
        Top(S,right);Pop(S); (* čtení ze zásobníku – reverzace pořadí *)
        Top(S,left); Pop(S);
        while left<right do begin (* vnitř. cyklus–dokud je co dělit *)
            Partition(A,left,right,i,j);
            Push(S,i); (* uložení intervalu pravé části do zásobníku *)
            Push(S,right);
            right:=j; (* příprava pravého indexu pro další cyklus *)
        end; (* while *)
    end (* while *)
end; (* procedure *)

```


Analýza Quicksortu.

- Quicksort patří mezi nejrychlejší algoritmy pro řazení polí.
- Quicksort je nestabilní a nepracuje přirozeně.
- Asymptotická časová složitost je lineární:
 - průměrná časová složitost je $TA(n) \sim 17 \cdot n \cdot \lg_2(n)$
 - časová složitost již seřazeného pole je:
 $T_{usp}(n) \approx 9 \cdot n \cdot \lg_2(n)$
 - časová složitost pro inverzně seřazené pole je:
 $T_{inv}(n) \sim 9 \cdot n \cdot \lg_2(n)$
- Dimenzování zásobníku nerekurzivního zápisu Quicksort:
 - Při uvedeném algoritmu, kdy se dělí vždy levý segment a pravý se uchovává, je třeba mít zásobník pro nejhorší případ t.j. , že se vždy segment rozdělí na jeden prvek a zbytek. Pak se musí uchovat $n-1$ dvojic indexů.

- Algoritmus, který bude dělit vždy menší segment a hranice většího uchová v zásobníku má nejhorší případ, když se interval vždy rozdělí na dva stejně velké segmenty. V tom případě je třeba zásobník dimenzovat na kapacitu $\lg_2 n$ dvojic indexů.
- Příklad: pokud by se řadilo pole o 1000 prvků, pak v případě prvního algoritmu je zapotřebí zásobník o kapacitě 999 dvojic. Ve druhém případě, kdy se menší segment dělí a větší uchovává, stačí zásobník o kapacitě $\lg_2 1000$, t.j. cca 10 dvojic.

Tabulka experimentálně naměřených hodnot pro Quicksort

n	256	512	1024
SP	10	24	50
OSP	22	48	50
ISP	12	26	56

K domácímu procvičení: Upravte uvedený algoritmus nerekurzivního zápisu Quicksortu pro variantu menšího zásobníku.

Řazení se snižujícím se přírůstkem – Shell-sort

- Rychlé algoritmy se vyznačují většími kroky prvků blížících ke svému správnému místu.
- Shell-sort je metoda, která ve své době vzbudila pozornost zvýšenou rychlostí a nepříliš průhledným principem. Shell sort pracoval na opakovaných průchodech podobných bublinovému řazení, kdy vyměňoval prvky vzdálené o stejný krok. Např. následující sekvence (reprezentovaná indexy) může být rozčleněna na čtyři podsekvence čísel vzdálených o krok 4:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	5	9	13	17															
2	6	10	14	18															
3	7	11	15	19															
4	8	12	16	20															

V první etapě je s použitím kroku 4 každá ze čtyř sekvencí zpracována jedním bublinovým průchodem. Ve druhé etapě se krok sníží na dvě a obě sekvence se zpracují jedním bublinovým průchodem. V poslední etapě se na celou sekvenci aplikuje bublinový průchod s krokem jedna. Tím je řazení ukončeno.

Teoretické analýzy nenašly nejvhodnější řadu snižujících se kroků (viz skriptu Vybrané kapitoly...). Kerningham a Richi (tvůrci jazyka C) publikovali verzi algoritmu, v níž první krok byl $(n \div 2)$ a v první etapě docházelo k výměně $(n \div 2)$ dvojic, pokud byly nebyly uspořádány v žádoucím směru. V další etapě se krok vždy půlil a n -tice zpracovávané bublinovým průchodem se zdvojnásobovaly. Poslední etapou byl průchod celým polem s krokem jedna.

```

procedure ShellSort(var A:TArr, N: integer);
var
    step,I,J:integer;
begin
    step:=N div 2; (* první krok je polovina délky pole *)
    while step > 0 do begin (* cykluj, pokud je krok větší než 1 *)
        for I:=step to N-1 do begin (* cykly pro paralelní n-tice *)
            J:=I-step+1;
            while (J>=1) and (A[J]>A[J+step])do begin (* bubl průchod *
                A[J]:=A[J+step];
                J:=J-step; (* snížení indexu o krok *)
            end; (* while *)
        end; (* for *)
        step:=step div 2; (* půlení kroku *)
    end; (* while *)
end; (* procedure *)

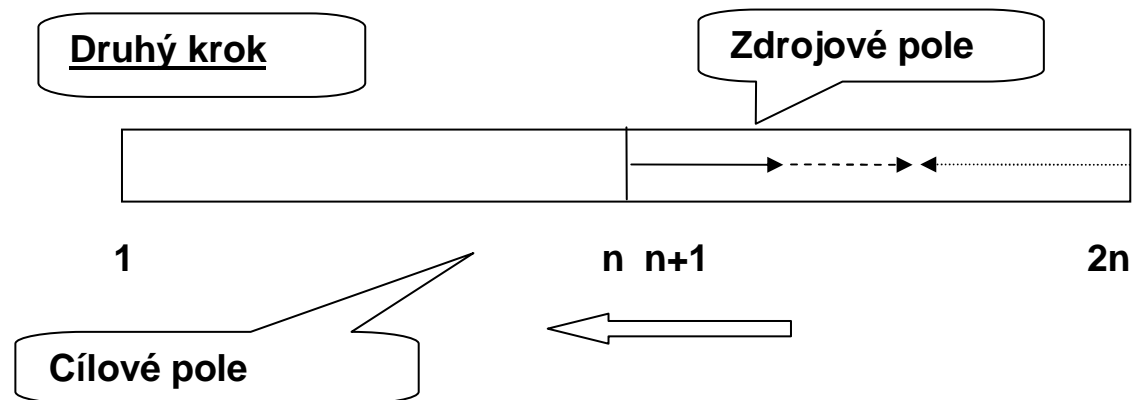
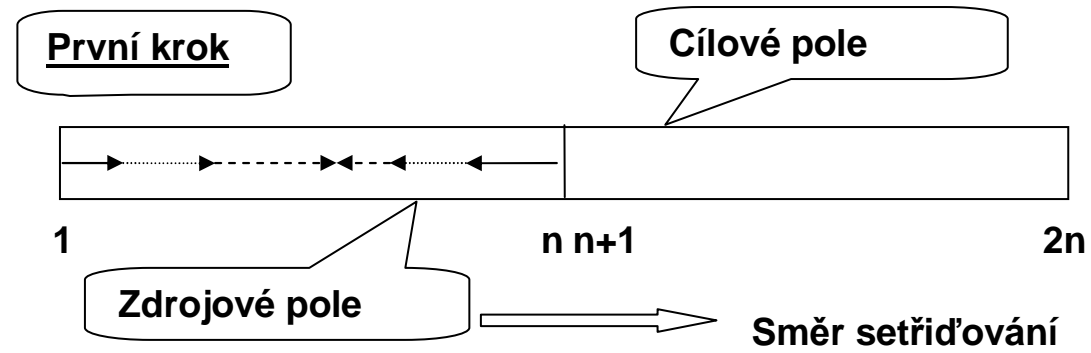
```

Hodnocení Shellsortu

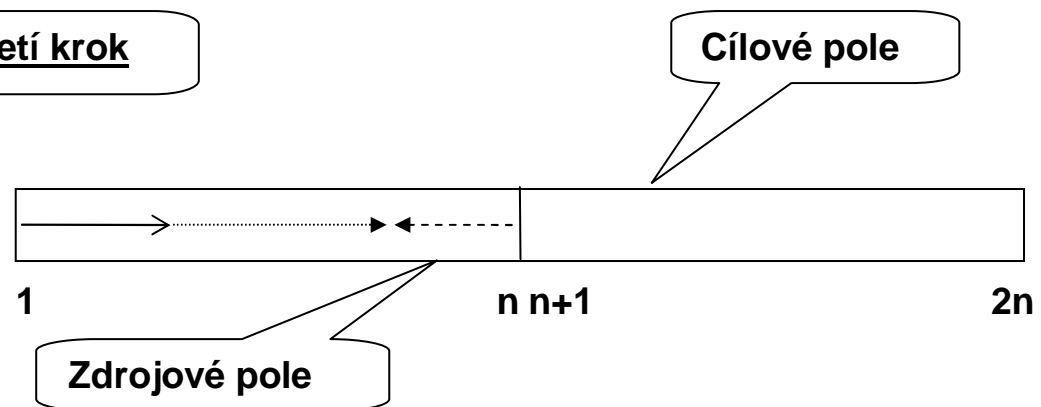
- Shell sort je nestabilní metoda. Pracuje „in situ“. V uvedené modifikaci pracuje rychleji než Heapsort ale pomaleji než Quicksort. Nepotřebuje ani rekurzi ani zásobník. Je nekorunovaným králem řadicích metod a zaslouží si mnohem větší pozornost, než nejznámější a mezi amatéry nejčastěji používané a přitom nejpomalejší bublinové řazení.

Řazení setřídováním - Merge sort

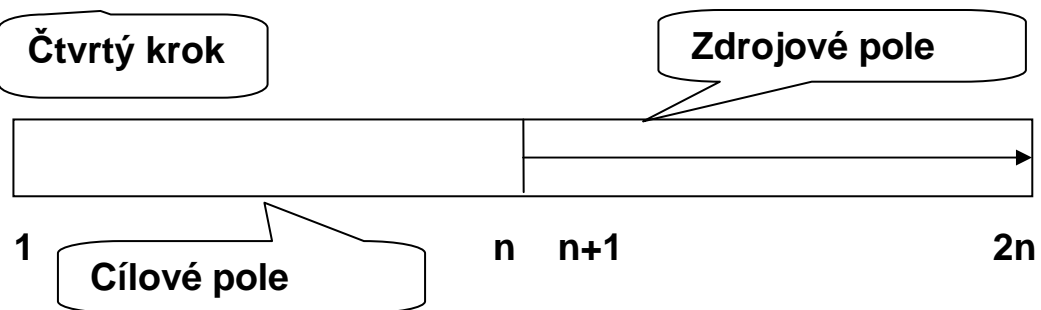
- Merge-sort je založen na principu slučování (setřídování), tedy na principu komplementárním k rozdělování.
- Merge-sort je sekvenční metoda využívající přímý přístup k prvkům pole. Postupuje polem zleva a současně zprava a setřídí dvě „proti sobě“ postupující neklesající posloupnosti. Výsledek se ukládá do cílového pole a počet vzniklých posloupností se počítá v počítadle. **Algoritmus končí, vznikne-li jen jedna cílová posloupnost.**
- Schéma postupu řazení Merge-sort je uvedeno na následujících obrázcích. Šipky zleva a zprava ve zdrojovém poli představují dvojice neklesajících posloupností, které jsou setříděny do jedné posloupnosti, která se uloží do cílového pole.



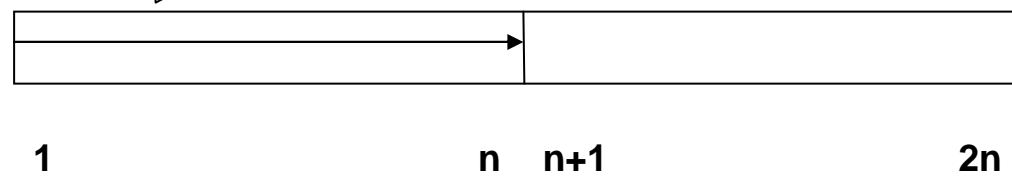
Třetí krok



Čtvrtý krok



Výsledek



Hodnocení Merge-sortu

- Algoritmus Merge-sortu je uveden ve skriptech „Vybrané metody...” a také v souboru 12TH_SES.RTF.
- Významným rysem algoritmu je jeho houpačkový mechanismus, který automaticky střídá pozici zdrojového a cílového pole i krok postupující proti sobě orientovanými slučovanými neklesajícími posloupnostmi. Tento mechanismus patří k základním programovacím technikám (viděli jsme ho již na verzi Bublínového řazení - „Shakersort”).
- Metoda Merge-sort je nestabilní, nechová se přirozeně a nepracuje „in situ”.

Asymptotická časová složitost je $TM(n) \sim (28 * n + 22) \lg_2(n)$


Algoritmus je velmi rychlý, ale z hlediska konstrukce programu nepatří k jednoduchým algoritmům. Z hlediska programovacích technik patří k velmi zajímavým algoritmům.

Experimentálně naměřené hodnoty jsou uvedeny v následující tabulce:

n	256	512
SP	8	13
NUP	6	12
ISP	32	72

Řazení polí setřídováním seznamů – Listmerge-sort

- Řazení polí setřídováním seznamů – ListMergeSort pracuje na principu slučování metodou bez přesunů položek. Proto je k základnímu poli nezbytné vytvořit stejně velké pomocné pole Pom indexových ukazatelů. Ty zřetězí neklesající posloupnosti.



Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	2	5	9	3	7	10	16	5	8	6	3	11	18	20
Pom	2	3	0	5	6	7	0	9	0	0	12	13	14	0

Prvním krokem algoritmu je zřetězení neklesajících posloupností do seznamu a vložení jejich začátků do seznamu začátků. V následujícím cyklu se vyzvednou ze seznamu začátky dvou zřetězených neklesajících posloupností. Jejich setříděním vznikne jedna zřetězená neklesající posloupnost, jejíž začátek se vloží na konec seznamu. **Algoritmus končí, je-li v seznamu již jen začátek jedné neklesající zřetězené posloupnosti.** Výsledek se může do podoby seřazeného pole zpracovat např. MacLarenovým algoritmem.

Jádrem algoritmu je setřídění dvou seznamů zřetězených v pomocném poli indexovými ukazateli. Schéma algoritmu je uvedeno ve studijní opoře.

Hodnocení ListMergeSortu

- ListMergeSort je algoritmus pracující bez přesunu položek. Je potenciálně stabilní (ve skriptech „Vybrané kapitoly... je nepřesně uveden, že je nestabilní). Stabilita se zajistí tím, že při setřídování se u shodných prvků musí do výstupní posloupnosti vložit prvek první posloupnosti.
- Experimentálně byly naměřeny hodnoty uvedené v následující tabulce.

n	256	512
SP	2	6
NUP	32	74
OSP	22	48

Řazení tříděním podle základu – Radix sort

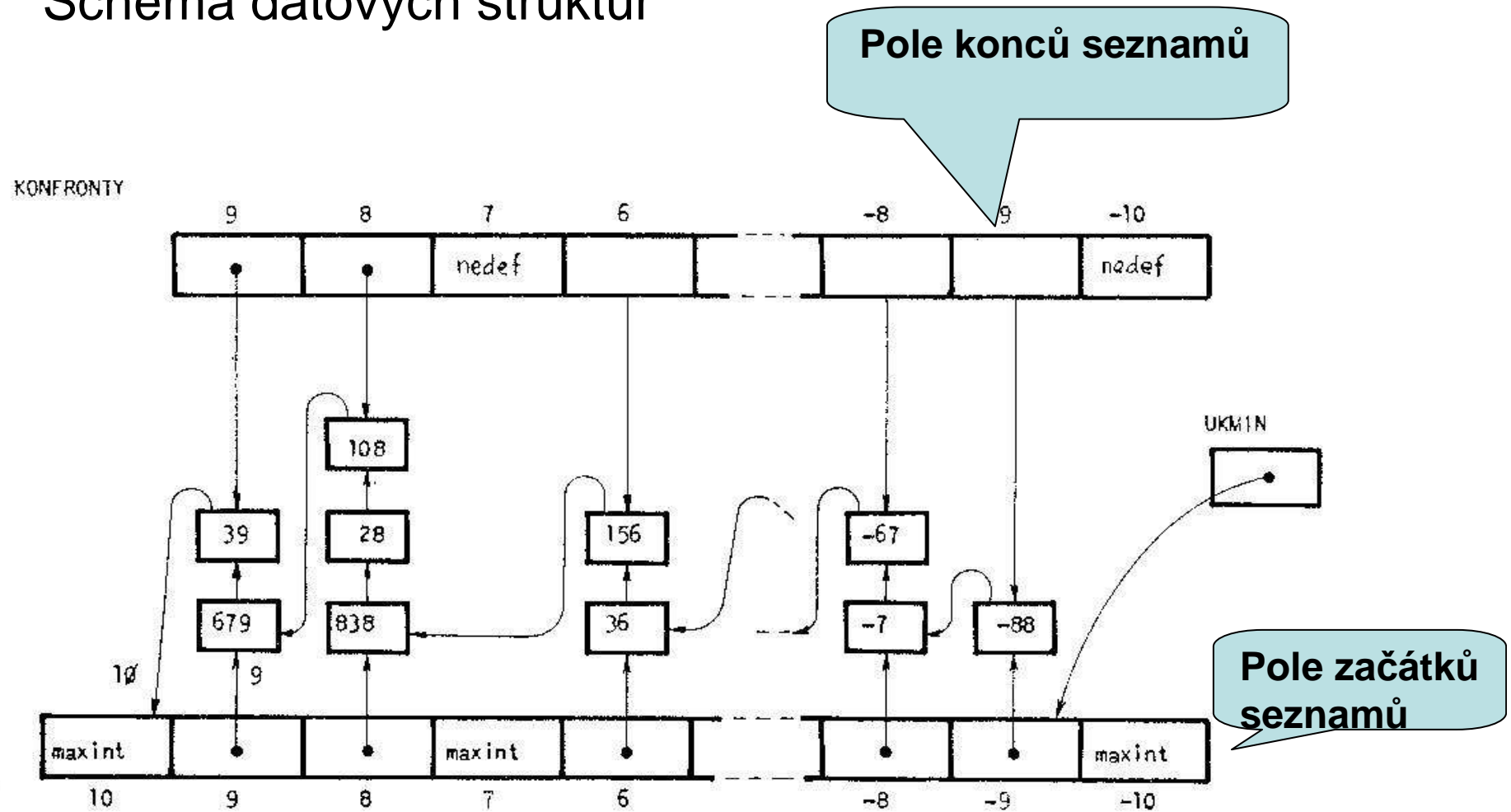
- Řazení tříděním podle základu je počítačová verze procesu řazení na děroštitkových třídících strojích. Na těchto strojích je základem desítková číslice na daném řádu v daném sloupci štitku. Ve většině počítačových aplikací je to desítková číslice čísla v kódu BCD(Binary-Coded-Decimal).
- Řazení tříděním je principiálně metodou pracující bez přesunu položek. Proto je třeba vytvořit k řazenému poli pomocné pole ukazatelových indexů.

Schéma postupu řazení trojciferných čísel

008	381	966	377	504	625	199	552	230	416	833	Pole před řazením
$\xrightarrow{\emptyset}$	$\xrightarrow{1}$	$\xrightarrow{2}$	$\xrightarrow{3}$	$\xrightarrow{4}$	$\xrightarrow{5}$	$\xrightarrow{6}$	$\xrightarrow{7}$	$\xrightarrow{8}$	$\xrightarrow{9}$		Třídění podle řádu \emptyset
$\xrightarrow{\emptyset}$	$\xrightarrow{1}$	$\xrightarrow{2}$	$\xrightarrow{3}$	$\xrightarrow{5}$	$\xrightarrow{6}$	$\xrightarrow{7}$	$\xrightarrow{8}$	$\xrightarrow{9}$			Třídění podle řádu 1
$\xrightarrow{\emptyset}$	$\xrightarrow{1}$	$\xrightarrow{2}$	$\xrightarrow{3}$	$\xrightarrow{4}$	$\xrightarrow{5}$	$\xrightarrow{6}$	$\xrightarrow{8}$	$\xrightarrow{9}$			Třídění podle řádu 2

Obr. 7.10. Řazení trojciferných čísel podle základu 10

Schéma datových struktur



Obr. 7.11. Datová struktura po sjednocení front vzniklých tříděním podle nejnižší číselice

Schéma algoritmu řazení RadixSort

begin

**Inicializace proměnných;
Stanovení hodnoty POCCIF – maximálního počtu číslic (míst)**

for j:=1 to POCCIF do begin

Inicializace pole ZACFRONTY;

Třídění do front podle j-té číslice

**Nalezení nejnižší neprázdné fronty (v poli ZACFRONTY) a
uložení jejího ukazatele do UKMIN**

Spojení front do jediného seznamu počínaje prvkem A[UKMIN]

end; (* for *)

Sekvenční seřazení prvků seznamu do výstupního pole

end;

Hodnocení Radix-Sortu

- Radix sort je stabilní metoda. Stav uspořádání nemá podstatný vliv na čas a proto se jeví jako by se nechoval přirozeně. Metoda nepracuje „in situ“, protože potřebuje pomocné pole indexových ukazatelů.

- Časová složitost má tvar:

$T_{MAX}(n) = (42 \cdot POCCIF + 15) \cdot n + (16 + 34 \cdot ZAKLAD) \cdot POCCIF + 15$, což jednoznačně indikuje lineární složitost! Teoreticky je to tedy nejrychlejší algoritmus.

Experimentálně byly naměřeny tyto hodnoty pro náhodně uspořádaném pole:

n	256	512	1024
NUP	24	60	102