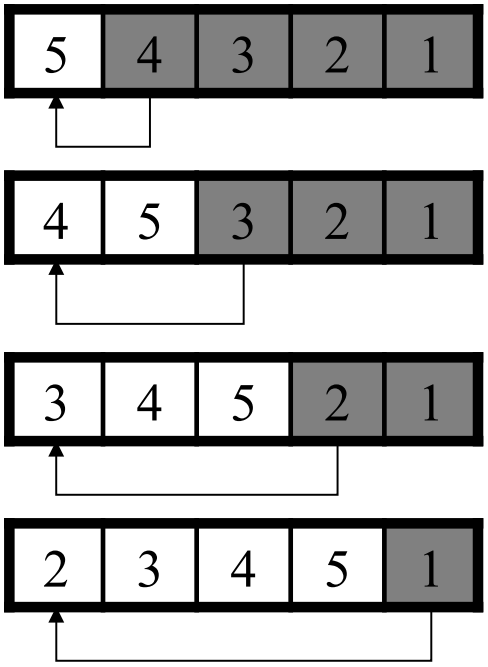
Složitost algoritmu

- Pojem „složitost“ algoritmu zahrnuje nejen časovou, ale také paměťovou náročnost algoritmu. Tímto problémem se zabývá jeden celý obor informatiky „Vyčíslitelnost a složitost“.
- Protože doba provádění i paměťové požadavky stejného algoritmu, implementovaného na různých strojích/překladačích může být díky různým HW/SW optimalizacím velmi různá, používá se ke zjišťování složitosti obecných algoritmů modelů výpočetních systémů. Nejčastěji používanými modely jsou:
  - **RAM** – Random Access Machine – počítač se definovanou sadou instrukcí, nekonečně velkou pamětí a vstupně/výstupní páskovou jednotkou
  - **Turingův stroj** (výpočetní prostředek s největší výpočetní mocností)
- Protože přesný algebraický výpočet složitosti pomocí implementace algoritmu např. na stroji RAM je velmi složitý už i u jednoduchých problémů, používají se pro srovnávání algoritmů častěji tzv. aproximatické odhady složitosti.

Aproximatické odhady složitosti

- Nepočítáme přesný počet taktů na provedení všech instrukcí, zaměříme se pouze na nejnáročnější operace, kterým přidělíme čas provedení=1, ostatní operace nás nezajímají-jejich čas provedení=0. Pak se pokusíme spočítat nejlepší a nejhorší případ doby běhu algoritmu pro různá vstupní data
- Příklady: třídění InsertSort



Počet přesunů: 2

Počet přesunů: 4

Počet přesunů: 6

Počet přesunů: 8

Celkem:

$$8+6+4+2 \approx 2\sum_{i=1}^{n-1} i = 2\frac{(n-1)^2}{2}$$

Funkce O, Ω, Θ

- V literatuře se aproximatická složitost vyjadřuje pomocí matematického aparátu - funkcí *O, Ω, Θ (omikron, omega, theta)*
- Nejčastěji se používá funkce *O* - např. formulace „časová složitost *s(n)* algoritmu insertSort je $O(k.n^2)$" nebo "$s(n)=O(k.n^2)$" znamená, že funkce časové složitosti InsertSortu *s(n)* roste maximálně tak rychle, jako funkce $f(n)=k.n^2$ (kde *k* je konstanta), tj. že existují konstanty $n_0 \geq 1$ a $c>0$ takové, že pro všechna $n \geq n_0$ a platí $s(n) \leq c.f(n)$
- Funkce Ω vyjadřuje, že nějaká funkce složitosti s(n) roste minimálně tak rychle jako funkce f(n), tedy:
  $s(n)= \Omega(f(n))$ pokud existují konstanty $n_0 \geq 1$ a $c>0$ takové, že pro všechna $n \geq n_0$ a platí $s(n) \geq c.f(n)$
- Funkce Θ vyjadřuje, že nějaká funkce složitosti s(n) roste řádově stejně rychle jako funkce f(n), tedy:
  $s(n)= \Theta(f(n))$ pokud platí současně $s(n)= \Omega(f(n))$ a $s(n)=O(f(n))$

Aproximatické odhady složitosti

- Kromě odhadů počtu přesunů je užitečné zabývat se i počtem porovnání - např. pokud budou řazené klíče typu řetězec, bude každé porovnání 2 klíčů na úrovni strojového kódu obnášet N porovnání znaků, kde N je délka řetězce kratšího klíče
- termín „Aproximatické odhady“ naznačuje, jakým způsobem nazírat na získané výsledky těchto odhadů - pomohou nám vybrat z množiny X možných algoritmů 2-3 vhodné.
- Pamatujte, že výsledná rychlost a paměťová náročnost nakonec velmi závisí také na:
  - efektivitě napsaného kódu
  - optimalizacích překladače
  - cílové platformě

Řazení

- řazení je proces seřazení údajů podle nějaké relace uspořádání
- v češtině se pro tento proces používají i další pojmy s lehkými významovými rozdíly:
  - třídění - tento pojem možná více odpovídá anglickému slovu „sort“ (quick-sort, bubble-sort), v češtině je na místě např. v případě algoritmu Radix sort, který vychází z principu třídění pomocí děrných štítků.
  - setřídění - je správné použít tento pojem v případě, kdy slučujeme několik seřazených sekvencí do jedné výsledné seřazené sekvence
  - uspořádání - matematický pojem relace uspořádání, což je funkce nebo operace, která definuje, který ze dvou prvků je „menší“
- v dějinách informatiky bylo vymyšleno mnoho algoritmů řazení a jejich modifikací, věčné slávě a publicitě se dostalo jen těm nejjednodušším a nejrychlejším

Kritérie výběru řadících algoritmů

- prostorová a časová složitost
- sekvenčnost: řadící algoritmus je sekvenční, pokud přistupuje k položkám řazené sekvence tak, jak jdou za sebou. Takový algoritmus pak lze použít např. k seřazení dat v lineárním seznamu
- přirozenost řazení: pokud je doba řazení seřazené sekvence < doba řazení náhodně uspořádané sekvence a ta je < doba řazení náhodně uspořádané sekvence, říkáme, že se algoritmus chová přirozeně
- stabilita: pokud algoritmus zachová vzájemné pořadí položek se stejným klíčem, říkáme, že je stabilní
- možnost paralelní implementace

Porovnání řadících algoritmů

Počet položek: 10, náhodně uspořádané pole, zdroj: sortchk, STL C++

Algoritmus	Počet porovnání	Počet přesunů
Bubblesort	42	72
Bubblesort (bidirectional)	47	72
Selectionsort	45	27
Insertionsort	30	42
Shellsort	30	42
Mergesort	22	68
Quicksort (Lomuto)	26	70
Quicksort (Randomized)	25	80
Quicksort (Hoare)	56	35
Heapsort	35	46

Porovnání řadících algoritmů

Počet položek: 100, náhodně uspořádané pole, zdroj: sortchk, STL C++

Algoritmus	**Počet porovnání**	**Počet přesunů**
Bubblesort	4892	7419
Bubblesort (bidirectional)	4039	7419
Selectionsort	4950	297
Insertionsort	2568	2671
Shellsort	779	1087
Mergesort	542	1344
Quicksort (Lomuto)	686	1412
Quicksort (Randomized)	646	1424
Quicksort (Hoare)	1071	568
Heapsort	384	434

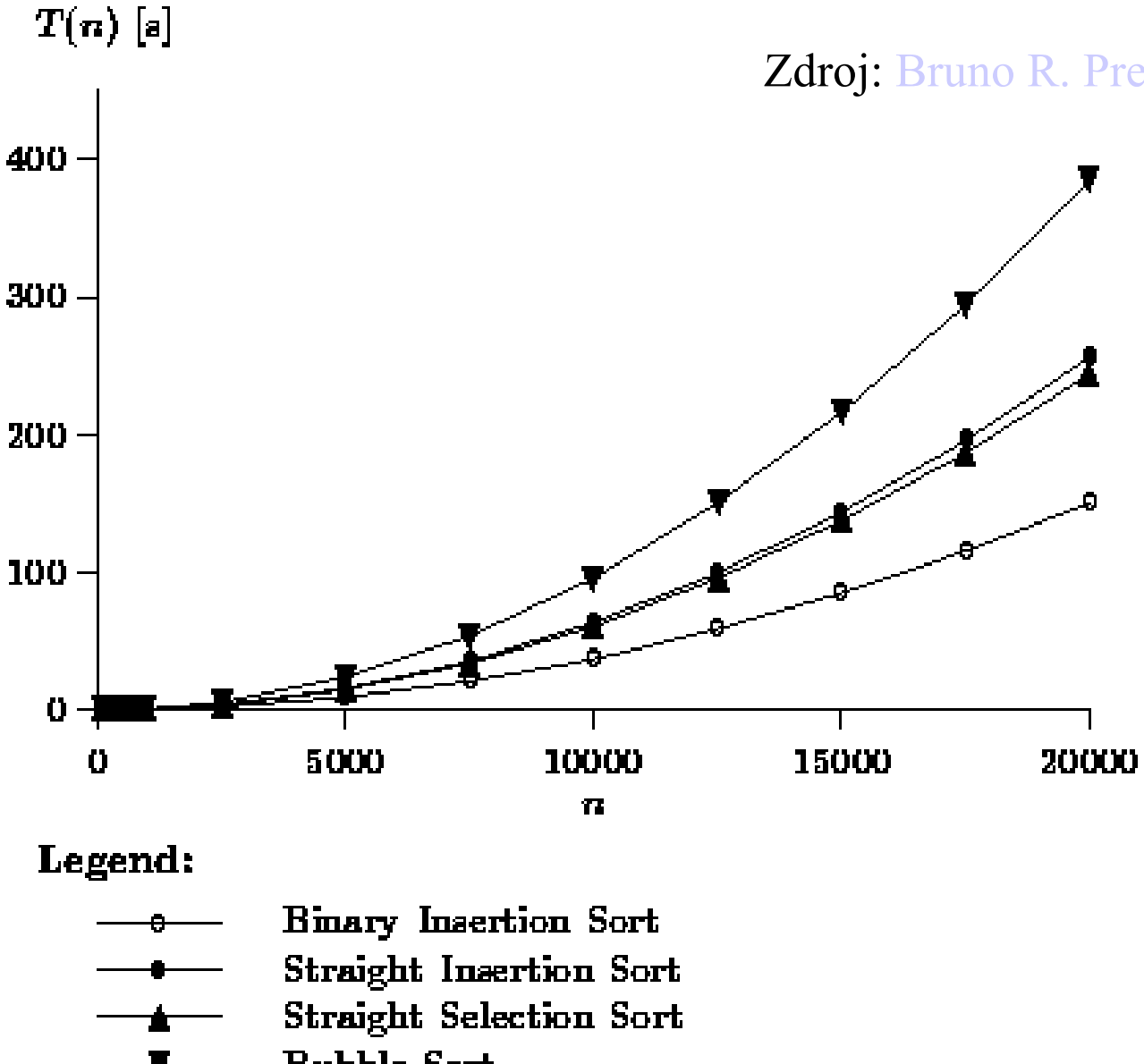
Porovnání řadících algoritmů

Počet položek: 10000, náhodně uspořádané pole, zdroj: sortchk, STL C++

Algoritmus	**Počet porovnání**	**Počet přesunů**
Bubblesort	49979584	74885447
Bubblesort (bidirectional)	37619489	74885447
Selectionsort	49995000	29997
Insertionsort	24971805	24981813
Shellsort	234228	313666
Mergesort	120457	267232
Quicksort (Lomuto)	152349	253604
Quicksort (Randomized)	156697	292643
Quicksort (Hoare)	208025	103398
Heapsort	37912	41876

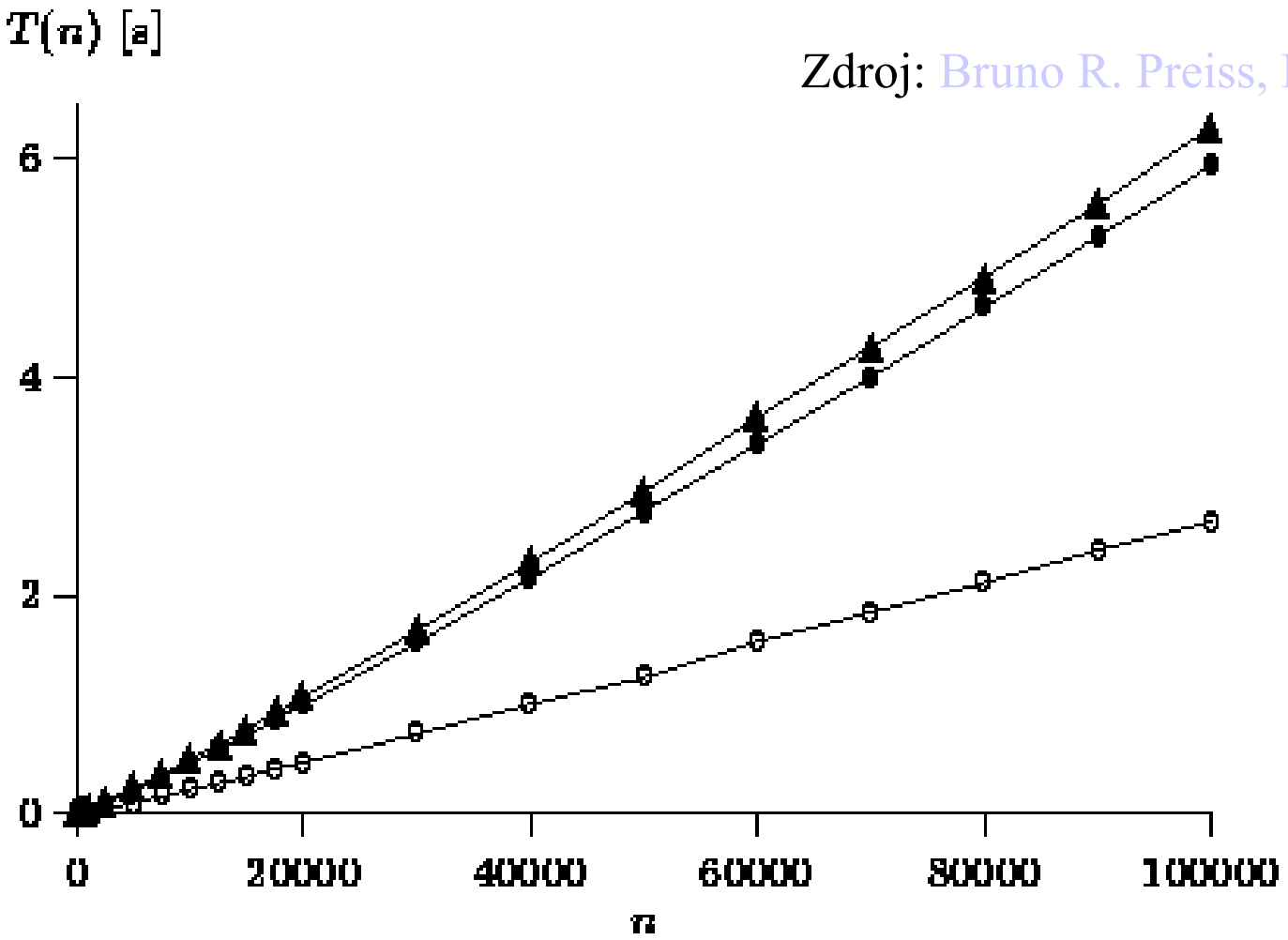
Doba běhu řadících algoritmů – SPARC 85MHz, náhodně uspořádané pole

Zdroj: Bruno R. Preiss, P.Eng.



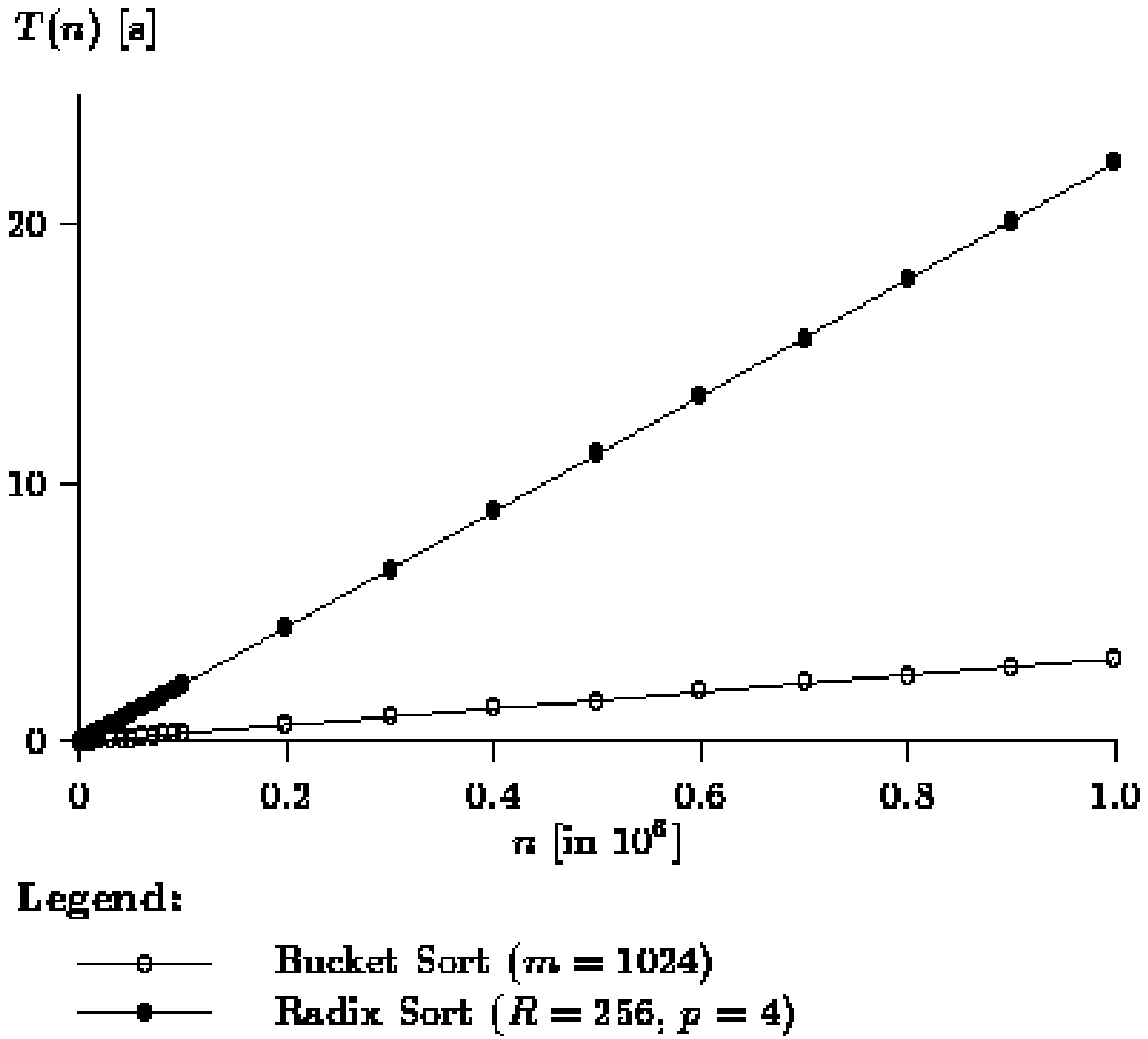
Doba běhu řadících algoritmů – SPARC 85MHz, náhodně uspořádané pole

Zdroj: Bruno R. Preiss, P.Eng.



Doba běhu řadících algoritmů – SPARC 85MHz, náhodně uspořádané pole

Zdroj: Bruno R. Preiss, P.Eng.



Bubble sort

Algoritmus ve 2 vnořených smyčkách prochází pole, porovnává vždy 2 sousední prvky, pokud nejsou ve správném pořadí, tak je prohodí – v každém průchodu tak největší prvek probublá na pravou stranu (konec) pole, v dalším průchodu tedy můžeme už procházet pouze n-1 prvků:

```
for i:=n downto 2 do { n=počet prvků pole }
      for j:=1 to i-1 do { probublávání úseku 1..i }
        if a[j]>a[j+1] then
          begin
            p:=a[j]; a[j]:=a[j+1]; a[j+1]:=p
              { výměna  prvků tak, aby měly správné pořadí}
          end;
```

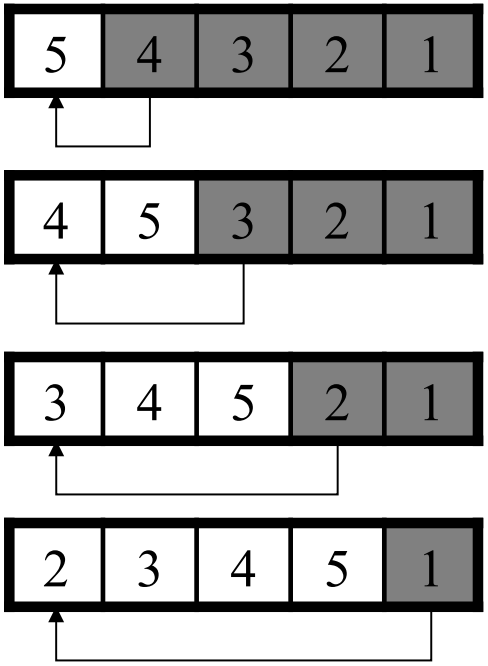
Bubble sort

- Algoritmus se chová stabilně a přirozeně, ale má ze všech algoritmů nejhorší časovou složitost $O(k.n^2)$
- Modifikace:
  - Přirozený bubble sort – pokud nedojde ve vnitřním cyklu k žádné výměně (tj. pole je seřazené), ukončí algoritmus. Toto je nejrychlejší algoritmus ze všech v případě, že je pole seřazené!
  - Ripple sort – pamatuje si pozici první dvojice, u které došlo k výměně, v příštím cyklu pak začne porovnávat až od předchozí dvojice
  - Shaker sort – prochází pole střídavě zprava a zleva
  - Shuttle sort – pokud dojde k výměně, pak vezme menší z vyměněných prvků a protlačuje jej doleva tak dlouho, dokud jej neumístí tam, kde patří

Straight Insert Sort

- Algoritmus vychází z předpokladu, že prvky levé části pole od indexu 1 do i-1 jsou již seřazené. Bere tedy postupně prvky z pravé části pole a zařazuje je na místo v levé části, kam patří:

```
for i:=2 to n do
     {1..i-1 je serazeny usek, a[i] do nej zarazujeme tak, aby zustal serazeny }
    begin
       j:=i-1; t:=a[i];
       while (j>0) and (a[j]>t) do
         begin { hledani vhodneho mista pro prvek a[i] }
           a[j+1]:=a[j]; dec(j)
         end;
       a[j+1]:=t;
     end;
```



- Algoritmus je stabilní a chová se přirozeně. Časová složitost je o něco lepší než u BubbleSortu

Binary Insert Sort

- Vychází ze Straight Insert Sortu s tím, že využívá skutečnosti, že levá půlka pole je již seřazená – je tedy možné pozici vkládaného prvku nalézt pomocí algoritmu binárního vyhledávání!
- Jedná se o nejrychlejší algoritmus ze třídy $O(n^2)$

```
for i:=2 to n do begin
  t:=a[i];
  low=1; high=i-1;
  while low<high do begin
    m:=(low+high) div 2;
    if x<a[m] then high:=m-1; else low:=m+1;
  end;
  for j:=i-1 downto low do a[j+1]:=a[j];
  a[low]:=x;
end;
```

- Metoda je stabilní a chová se přirozeně

Quick Sort

- Quick sort pracuje na principu „Rozděl a panuj“
- Quick sort se nechová stabilně ani přirozeně
- **QuickSort není nejrychlejší řadící algoritmus!!** Průměrná časová složitost pro náhodně uspořádané pole je $N.\log_2 N$, ale pro špatně uspořádanou posloupnost **může degradovat až na N^2!**
- QuickSort používá rekurzi = skrytá paměťová složitost! Pokud časová složitost degraduje k N^2, pak zároveň spotřeba paměti na zásobníku bude k.N! (k=počet bajtů pro uložení parametrů, návr. adresy..)
- Quicksort funguje tak, že rekurzivně každou část pole dělí na 2 části tak, aby v levé části byly všechny prvky menší než v pravé
- Základ algoritmu vypadá takto:

```
procedure quicksort(levy,pravy:integer);
    var pivot:integer;
    begin
      if levy<pravy then { aspoň 2 prvky }
        begin
          pivot=rozdel(levy,pravy);
          if levy<pivot-1 then quicksort(levy,pivot-1);
          if pravy>pivot+1 quicksort(pivot+1,pravy);
        end;
end;
```

- Jádro algoritmu spočívá v proceduře rozděl, která musí zadanou část pole rozdělit na pravou a levou část (ideálně půlky), a uspořádat v ní prvky tak, aby všechny prvky v levé části byly menší než prvky v pravé části
- Na volbě algoritmu procedury rozdel závisí efektivita celého algoritmu! Nejznámější algoritmy pocházejí od pana Hoareho a pana Lomuta:

Quick Sort – Hoareho varianta

- Procedura *rozdel* musí najít vhodný prvek – ideálně medián, který by se v seřazené posloupnosti nacházel v její polovině. Protože nalezení opravdového mediánu by bylo časově náročné, zvolíme si místo něj prvek v půlce aktuální části pole. Pak procházíme pole zleva i zprava. Pokaždé, když v levé části najdeme prvek, který do ní nepatří (je větší než pivot), zastavíme se na něm a jdeme hledat prvek v pravé části, který naopak nepatří do pravé části (je menší než pivot). Když takový najdeme, prohodíme oba nepatřičné prvky:

```
function rozdelHoare(levy, pravy : integer):integer;
  var  pivot : Integer;
  begin
    pivot := p[(levy + pravy) div 2];
    repeat
      while (p[levy] < pivot) do levy := levy + 1;
      while (p[pravy] > pivot) do pravy := pravy - 1;
      if (levy <= pravy) then begin
          vymen(pravy, levy);
          pravy := pravy - 1;
          levy := levy + 1
        end;
    until levy > pravy;
    pivot:=levy;
  end;
```

Quick Sort – varianta Lomuto

- Nico Lomuto (z firmy Alsys, Inc.) v roce 1984 publikoval variantu, která pro náhodně uspořádané pole vykazuje menší počet porovnání a větší počet přesunů, než Hoareho varianta. Vhodné použití je např. pro použití řazení klíčů typu řetězec, protože porovnání řetězců je časově náročné, ale přesun se dá řešit přesunem ukazatelů.
- Jediný rozdíl je v proceduře *rozdel*, která zařazuje prvky, menší než pivot, do levé půlky pole, která postupně narůstá doprava. Pokud se jako pivot volí hned první prvek, pak v případě, že tento prvek bude pokaždé nejmenším prvkem ze všech, degraduje metoda na složitost $O(n^2)$. Proto se jako první prvek dá vybírat náhodný prvek.

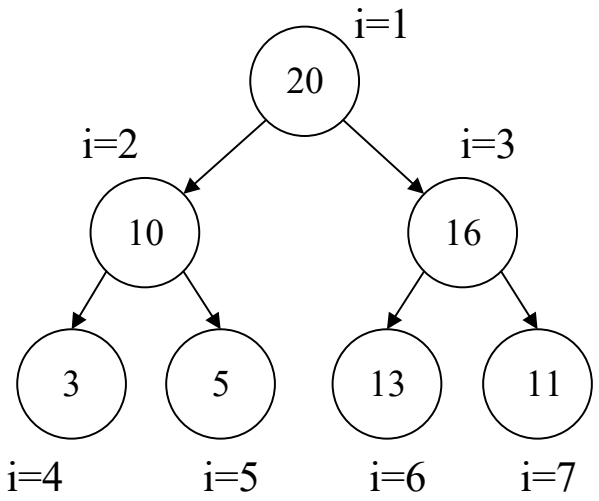
```
procedure rozdel(levy, pravy:integer; var ip:integer);
    var pivot:integer;
       i,pmin:integer;
    begin
      pivot:=p[levy]; pmin:=levy;   { pivot je první prvek, pmin je jeho index }
      for i:=levy+1 to pravy do      { projdi všechny prvky mezi levým a pravým  }
        if p[i]<pivot then begin      { aktuální prvek patří do levé části? }
          inc(pmin); vymen(pmin,i);   {ano? zvyš index konce levé části, která }
        end;                          {obsahuje prvky<pivot, a ulož tam tento prvek}
      vymen(levy,pmin);          { Na konec zbývá dát pivot na správné místo}
      rozdel:=pmin               { vráť index, na kterém je umístěný pivot }
    end;
procedure vymen(i,j:integer);
    var x:integer;
    begin x:=p[i]; p[i]:=p[j]; p[j]:=x; end;
```

Heap sort

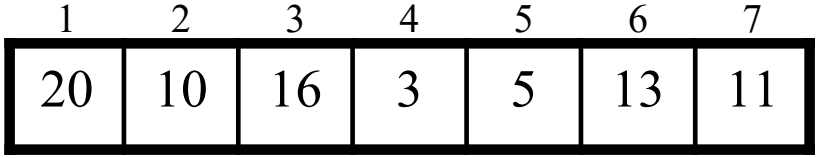
- Jeden z nejrychlejších algoritmů, pracujících v in-site (bez potřeby další paměti!)
- Heapsort je nestabilní a nechová se přirozeně
- Princip je velmi podobný insert sortu
  - pole je rozděleno na levou neseřazenou část a seřazenou část vpravo
  - V neseřazené části najdeme největší prvek a přidáme jej k seřazené posloupnosti zleva
  - Oproti binárnímu insertsortu je rychlejší v tom, že není potřeba „vytlačovat“ prvky za vkládaným prvkem na správnou pozici

Heap sort

- Rychlost hledání maxima v neseřazené části je dosažena díky velmi sofistikované metodě rozvržení prvků v poli – celé pole je totiž interpretováno jako speciální stromová struktura - tzv. heap, ve které platí (pro vzestupné řazení):
  - Otec je větší, než oba jeho synové. Kořen tedy obsahuje maximum.
  - Pokud je index otce = i, pak levý syn má index 2*i, pravý syn 2*i+1. Uspořádání synů je libovolné (pravý syn může být větší nebo menší než levý)
  - Z logiky stromu vyplývá, že v první půlce pole budou otcové, ve 2.půlce budou listy
  - Příklad:



~



Heap sort

```
procedure HeapSort(var data:array[1..N] of integer; pocetPrvku : Integer);
 var indexUzlu : Integer;
 begin                                                    {počáteční ustavení hromady:}
  for indexUzlu := (pocetPrvku div 2) downto 1 do {ber všechny otce od spodního patra}
    ustavHromadu(indexUzlu, pocetPrvku);          {a zařiď, aby splňovali pravidla hromady}
                {v prvním prvku pole (kořenu haldy) máme nyní maximum.}
                {Rozdělíme si teď pole na neseřazenou část (vlevo) a seřazenou část (vpravo) }
  for indexUzlu := pocetPrvku downto 2 do begin   {seřazená část roste od konce pole}
      Swap(data[1], data[indexUzlu]);       { vlož maximum na začátek seřazené části vpravo}
      ustavHromadu(1, indexUzlu - 1);   { a ve zmenšené neseřazené části}
                                                  { najdi nové maximum, a ulož jej do kořene (index 1)}
    end
 end;
```

Heap sort

```
procedure ustavHromadu(Root, max_index:Integer);
  var HeapOk : Boolean;
    levySyn, vetsiSyn : Integer;
  begin
    levySyn:=Root*2; HeapOk := False;
    while (levySyn <= max_index) and not HeapOk do begin
       if (levySyn = max_index) then vetsiSyn := levySyn        {najdi většího ze synů uzlu}
       else if (data[levySyn] > data[levySyn + 1]) then vetsiSyn := levySyn
       else vetsiSyn := levySyn + 1;
       if (data[Root] < data[vetsiSyn]) then begin {otec je < větší syn?}
          Swap(data[Root], data[vetsiSyn]);         {prohoď je, aby splňovali pravidla hromady}
          Root := vetsiSyn;                         {a s bývalým otcem pokračuj v testování}
          levySyn := 2 * Root;                      {v nižších patrech stromu}
       end else HeapOk := True                      {otec je > syn = konec…}
      End
  End;
```

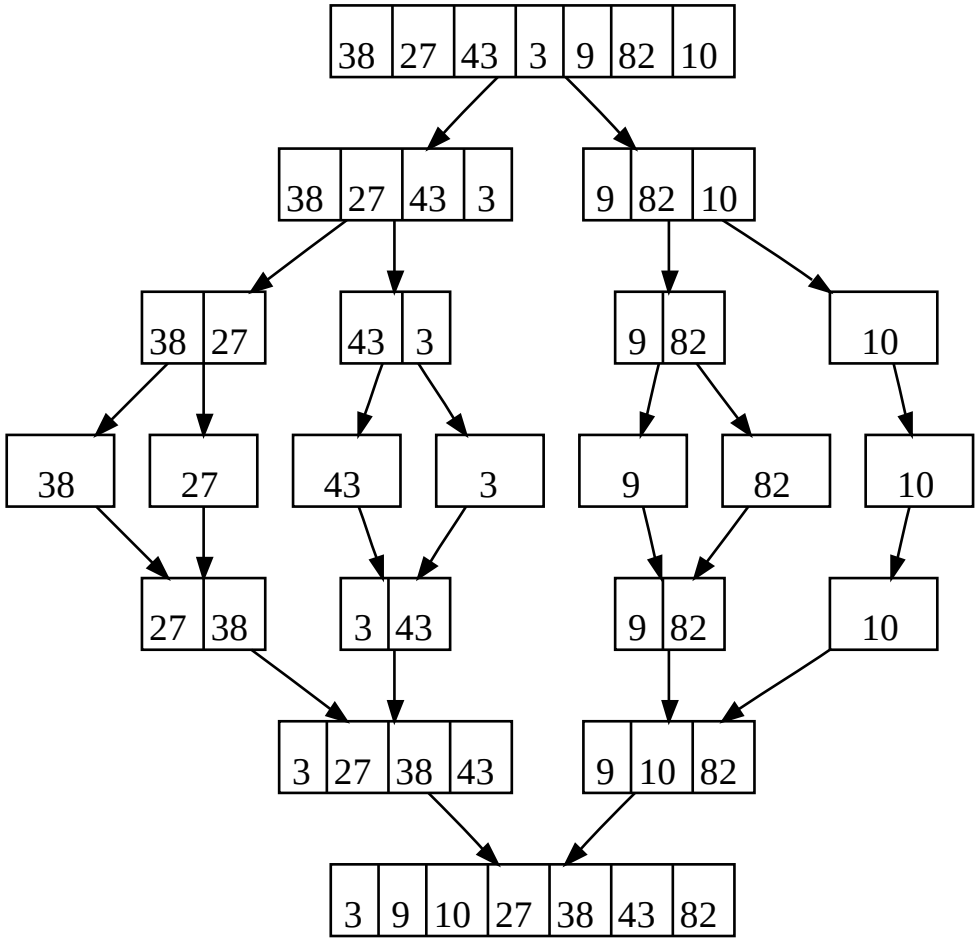
Animace Heapsortu

- http://www.research.compaq.com/SRC/JCAT/jdk10/sorting/
- http://www.cs.okstate.edu/~jonyer/teaching/5413/applets/sort1/heapsort.html
- http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html
- http://www.sci.brooklyn.cuny.edu/~gurwitz/bookmarks/alg.html - animace dalších algoritmů

Merge Sort

- Autor: John Von Neumann, 1945
- Metoda „Rozděl a panuj“, časová složitost $O(n.\log_2 n)$
- Stabilní
- Jednoduchá paralelní implementace
- Paměťová složitost O(n)
- Sekvenční přístup = lze řadit soubory i seznamy.
- Díky tomu je použit jako defaultní řadící algoritmus ve většině vyšších programovacích jazyků.

Princip funkce



Zdroj obrázku: http://en.wikipedia.org/wiki/Merge_sort

Implementace v C

```
typedef int tData;

void merge(tData *arr, int low, int mid, int high) {
  int left=low, right=mid+1, i=0, rest, size=high-low+1;
  tData res[size];

  while (left <= mid && right <= high)
    if (arr[left] <= arr[right])          //move smaller numbers to the result array
      res[i++] = arr[left++];
    else
      res[i++] = arr[right++];
    if (left > mid) rest=right;          //part of left or right half was not moved completely,
      else rest=left;                                 //so me move it now:
    while (i < size) res[i++] = arr[rest++];       //@todo: toto lze zefektivnit=spojit
    for (i = 0, left=low; i < size; i++, left++)  //s kopírováním res do arr
      arr[left] = res[i];                //finally, copy the result back to the array
}

void merge_sort(tData *array, int low, int high) {
  int mid;
  if (low < high) {
    mid = (low + high) / 2;
    merge_sort(array, low, mid);
    merge_sort(array, mid + 1, high);
    merge(array, low, mid, high);
  }
}
```

Implementace bez rekurze

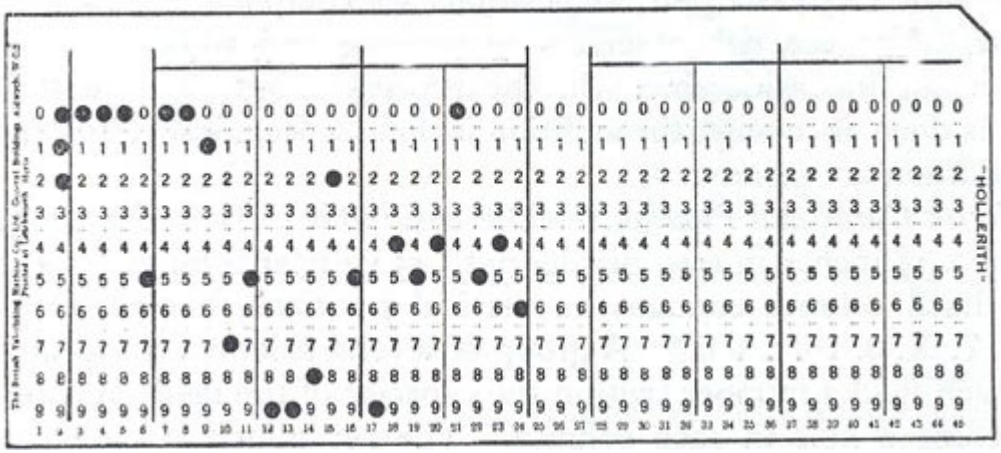
```
void mergeSort(int arr[], int n)
{
  int curr_size;  // For current size of subarrays to be merged from 1 to n/2
  int left_start; // For picking starting index of left subarray to be merged

  for (curr_size=1; curr_size<=n-1; curr_size = 2*curr_size)
  {
    // Pick starting point of different subarrays of current size
    for (left_start=0; left_start<n-1; left_start += 2*curr_size)
    {
      // Find ending point of left subarray. mid+1 is starting point of right
      int mid = min(left_start + curr_size - 1, n-1);

      int right_end = min(left_start + 2*curr_size - 1, n-1);

      merge(arr, left_start, mid, right_end);
    }
  }
}
```

Příklad použití

```
int main(int argc, char * argv[]) {
    tData example[]={10,9,8,7,6,5,4,3};
    merge_sort(example, 0, sizeof(example)/sizeof(tData)-1);
    return EXIT_SUCCESS;
}
```

Radix sort

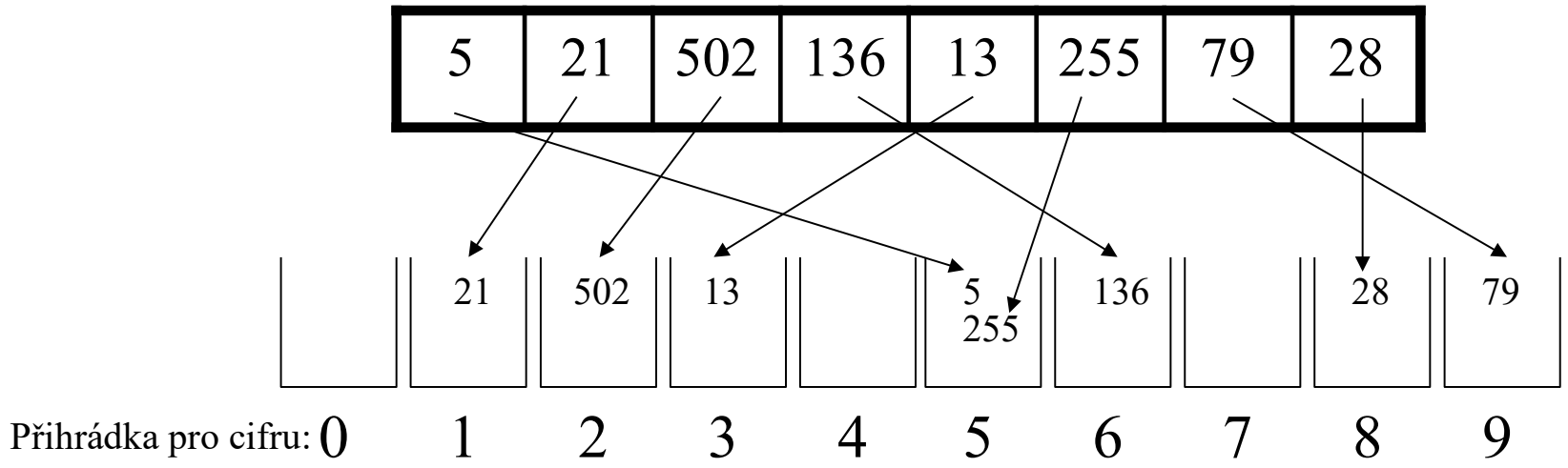
- Super rychlý algoritmus s časovou složitostí = O(N)!!!
- Za rychlost ovšem platí:
  - Spotřebou paměti – vyžaduje minimálně stejně velkou paměť, jako zabírá tříděné pole
  - Umí třídit pouze čísla typu integer
- Princip vychází z třídění děrných štítků:
  - Děrný štítek měl užitečná data zapsána ve sloupcích, každý sloupec představoval jednu cifru v dekadické soustavě
  - Řazení se provádělo tříděním štítků podle cifer v jednotlivých sloupcích od nejnižšího řádu – podle jednoho sloupce pak štítky propadly do 10 šuplíků, z nichž je operátor vzal, spojil za sebe, a třídění pokračovalo podle dalšího řádu (sloupce)



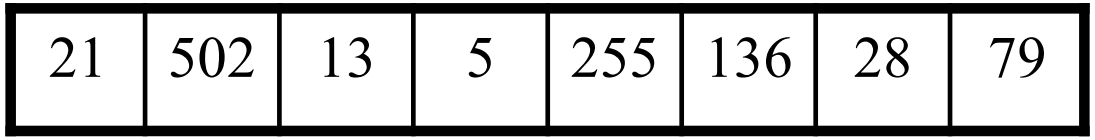
Radix sort

Algoritmus je tedy následující:

- Bereme postupně prvky pole a třídíme je do přihrádek podle cifry v aktuálním řádu. Začínáme nejnižším řádem (jednotky), pokračujeme desítkami, stovkami atd:

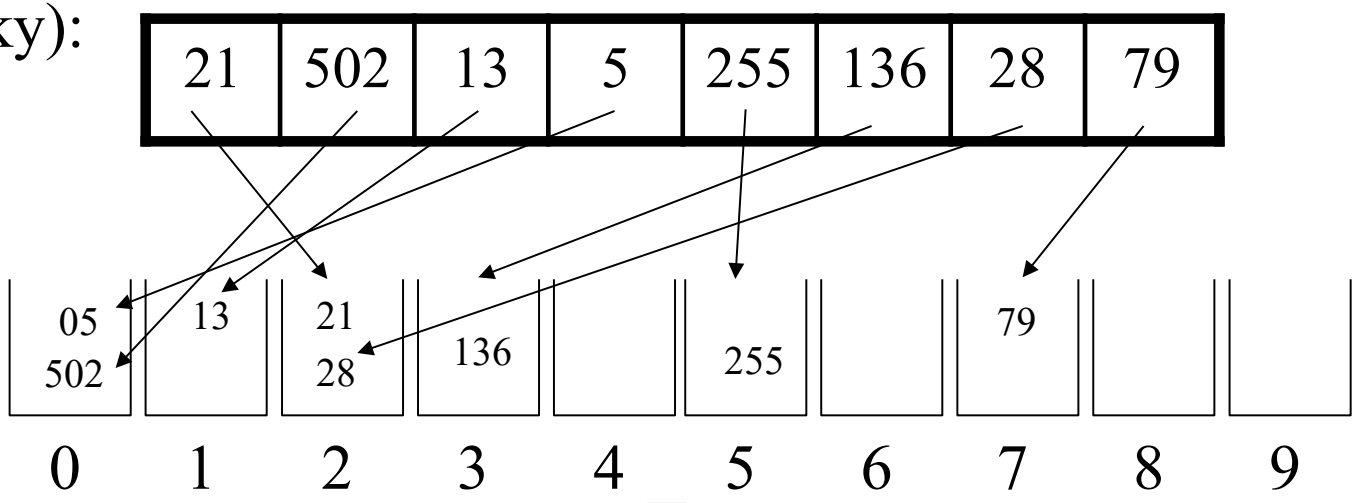


- Po zatřídění do přihrádek poskládáme čísla z přihrádek zpět do pole – začneme přihrádkou 0, pokračujeme přihrádkou 1 atd.:

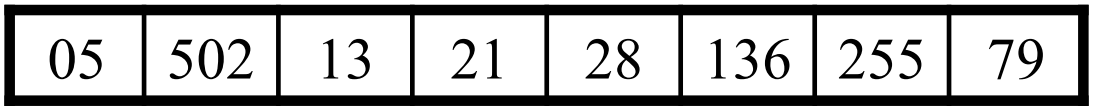


Radix sort

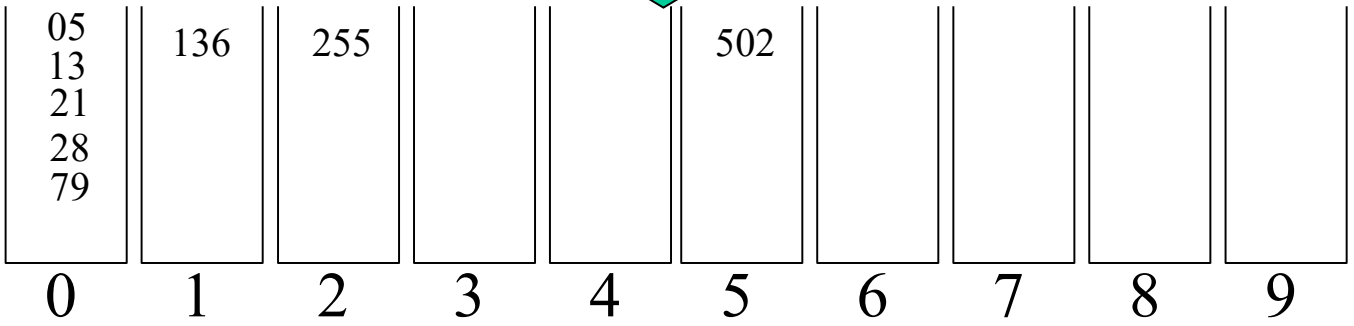
- Pokračujeme čísel v poli do přihrádek podle číslic v dalším řádu (desítky):



Poskládání přihrádek do pole:

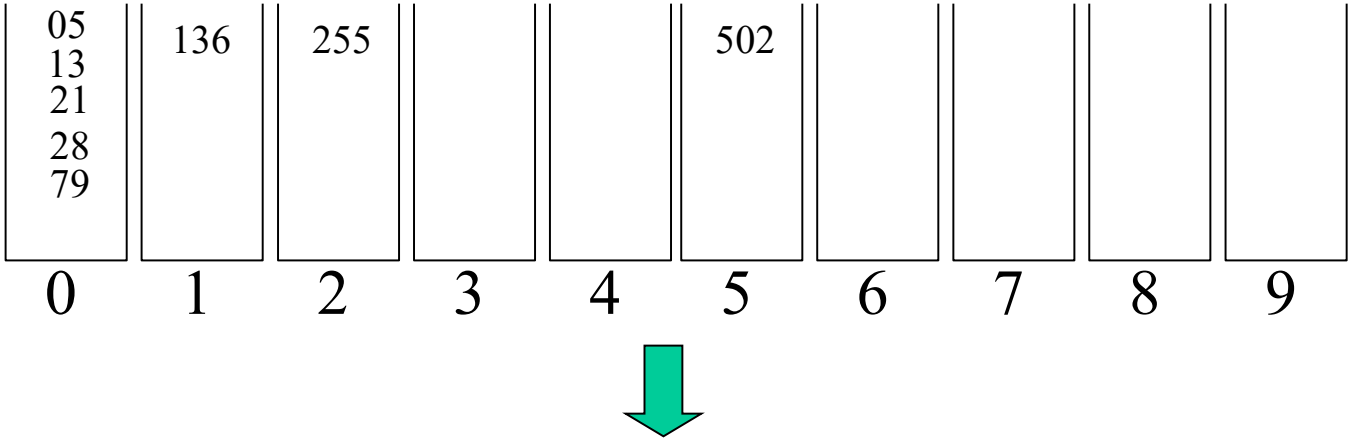


Třídění dle řádu 10^2:



Radix sort

- Pokračujeme čísel v poli do přihrádek podle číslic v dalším řádu (desítky):



Poskládání přihrádek do pole:

05	13	21	28	79	136	255	502

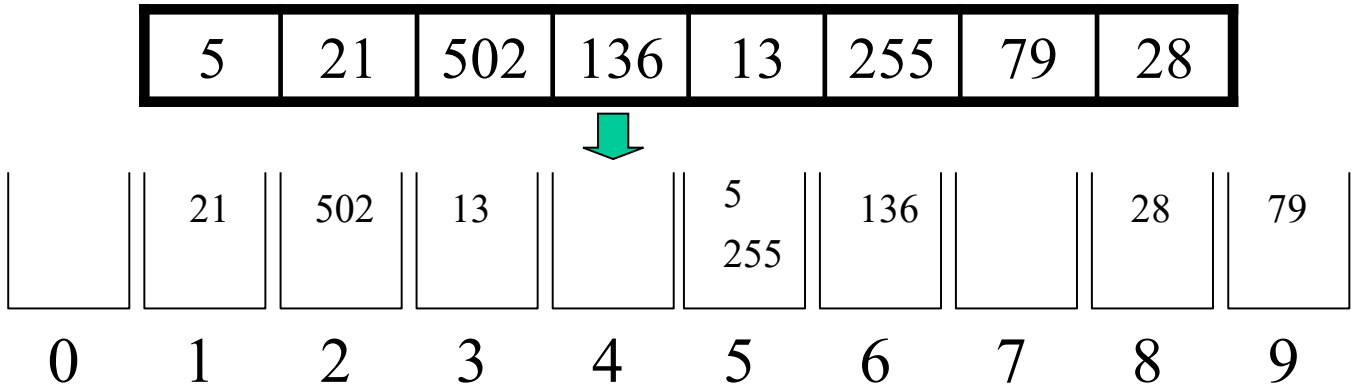
- Přihrádky implementujeme jednoduše pomocí ADT Fronta (Queue). Výsledný algoritmus by pak vypadal například takto:

Radix Sort

```
procedure RadixSort (Var A : ArrayType);
var Queue : tQueue;
  I, J, delitel, maximum   : Integer;
begin
  maximum:=max(A);
  for I := 0 to 9 do InitQueue(Queue[I]);
  delitel := 1;
  while delitel <= maximum do
  begin
    I := 1;
    for I:=1 to N do begin
      InsertQueue (Queue[Number div delitel MOD 10], A [I]);
      I := I + 1;
    end;
    J:=1;
    for I:=0 to 9 do
      while not QueueEmpty(Queue[I]) do begin
        A[J]:=PopQueue(Queue[I]); J:=J+1;
      end;
    end;
    divisor := 10 * divisor;
  end;
end;
```

Radix sort – jak na fronty?

- Místo fronty realizované pomocí lineárního seznamu s ukazateli můžeme použít paměťově úspornější implementaci front ve statickém poli, kde využijeme faktu, že celkový počet položek ve všech frontách je = N (kde N je počet prvků řazeného pole)
- Podívejme se na stav front po prvním kroku Radix sortu z předchozího příkladu:



- Tyto fronty je možné realizovat pomocí principu zřetězených seznamů ve statickém poli – pak nám pro realizaci všech front stačí jediné statického pole P o N prvcích, ke kterému budeme přistupovat tak, že P[i] bude indexem na následníka prvku data[i] v nějaké frontě. Pokud data[i] nemá následníka, bude P[i]=maxint. Každá z front 0..9 bude pak realizovaná pomocí dalších proměnných - indexů začátku a konce fronty v poli data. Pro uvedený příklad pole data bude obsah pole P vypadat takto:

6 (index 255, které je za 5)	**maxint** (za 21 není ve frontě nic…)	**maxint** (za 502 není ve frontě nic…)	**maxint** (za 136 není ve frontě nic…)	**maxint** (za 13 není ve frontě nic…)	**maxint** (za 255 není ve frontě nic…)	**maxint** (za 79 není ve frontě nic…)	**maxint** (za 28 není ve frontě nic…)

Radix sort – jak na fronty?

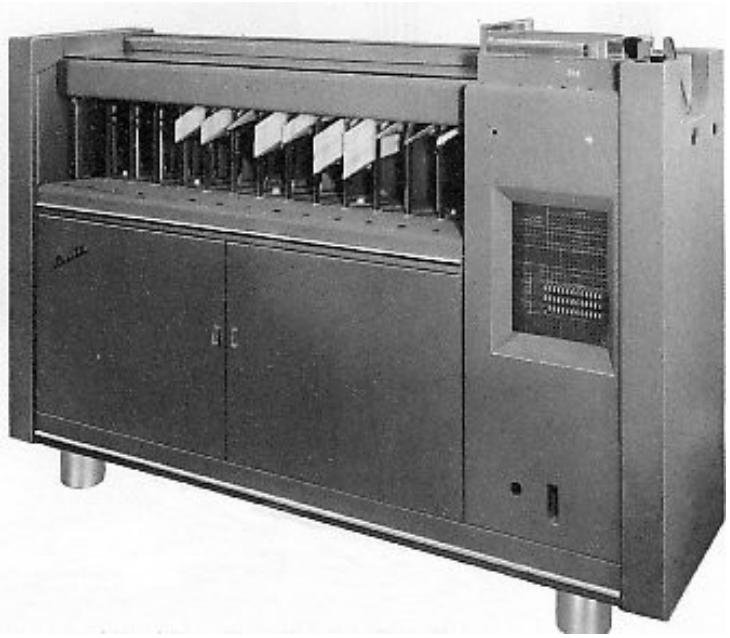
- Kromě pole P budeme tedy potřebovat ještě indexy začátků a konců všech 10ti front...

Radix Sort

- Pokud je potřeba řadit i záporná čísla, je potřeba přidat dalších 9 front i pro záporné číslice
- Velkou optimalizací je použití základu = 16, protože v 16kové soustavě operaci modulo 16 realizujeme bitovou operací (cislo AND 15), a operaci dělení 16 zase operací (cislo SHIFT_RIGHT 4)

Radix sort

- Třídička děrných štítků IBM z roku 1925, na obrázku vedle ní poslední model z 60tých let, který třídil až 2000 štítků/min



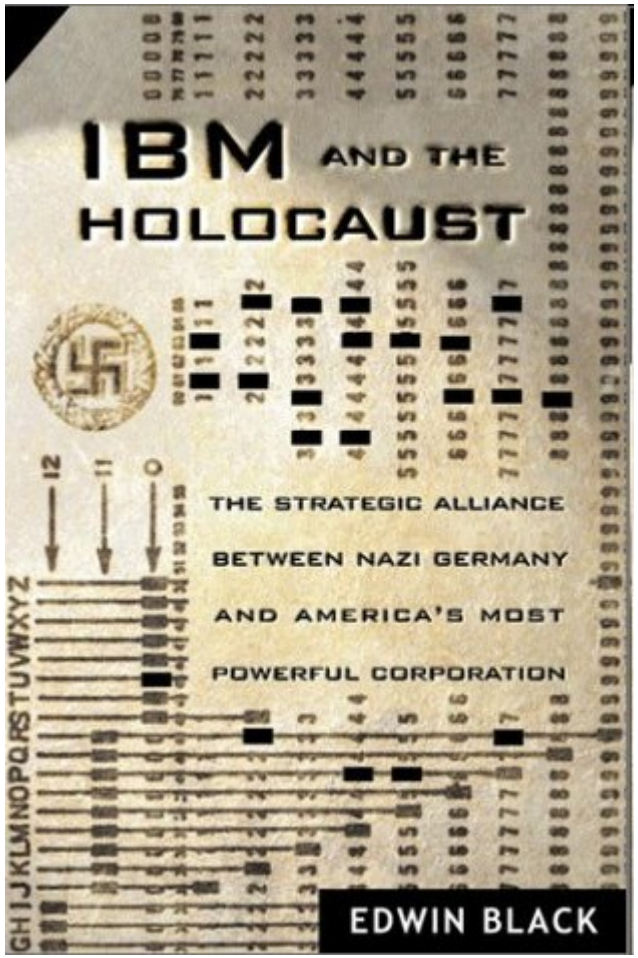
Programování a morálka

Co má společného řazení děrných štítků

a Holocaust?

Kde najít další informace

- Edwin Black: IBM and the Holocaust, 2001



Bucket (bin) sort

- Radix sort, který nepracuje podle jednotlivých řádů v čísle, ale pro každou možnou hodnotu v řazeném poli má speciální šuplík.
- Díky tomu, že nemá K průchodů jako Radix sort pro K řádů, je K-krát rychlejší než radix sort
- Dá se použít jen v případě pokud řadíme pole s omezenou množinou hodnot – spotřeba paměti je totiž jinak extrémní!

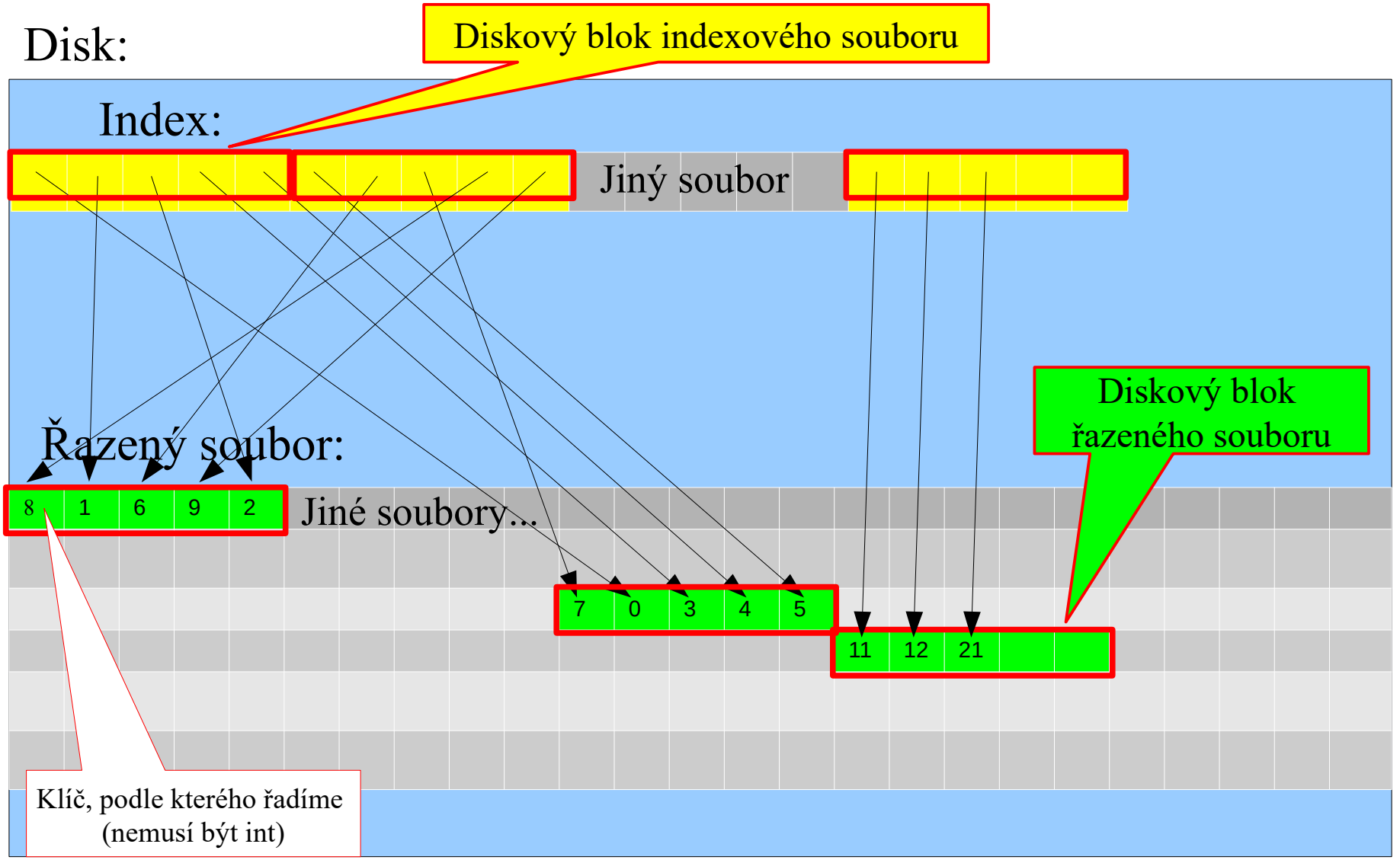
Řazení pomocí indexů

- Pro velká pole nebo soubory, obsahující prvky (záznamy) s velkou velikostí, nebudeme přesouvat celé záznamy, ale pouze ukazatele na ně.
- U souborů musí být takovéto ukazatele uloženy v poli, obsahujícím indexy jednotlivých záznamů – pak se neřadí fyzicky celý soubor, ale pouze pole indexů.
- V databázových systémech, které obsahují miliony položek, se tato pole ukládají na disk (tzv. indexové soubory), aby se nemusel algoritmus řazení provádět při každé operaci, vyžadující seřazená data.

Indexy v databázích na disku

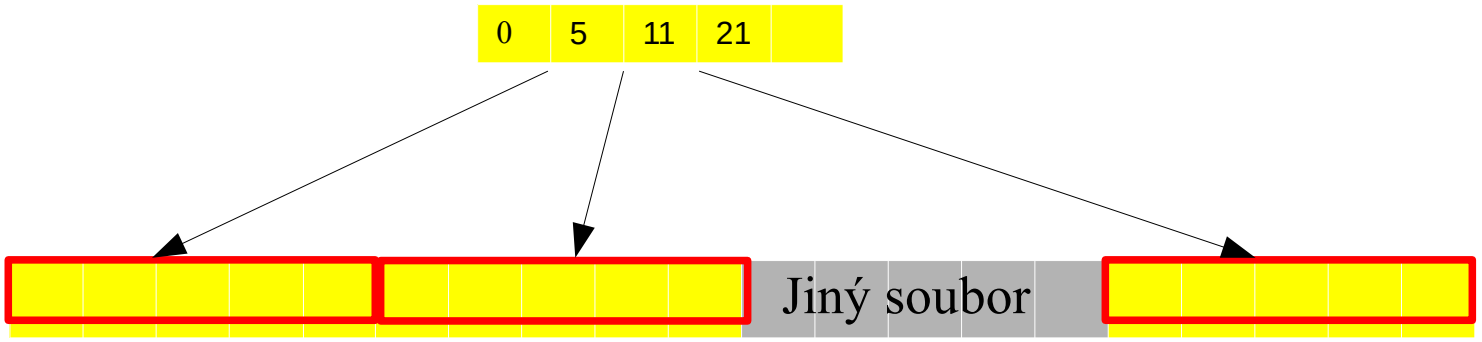
- Indexy zvyšují výkon vyhledávání v neseřazených sloupcích, tzn. i operací JOIN, díky možnosti použití algoritmu hledání metodou půlení intervalu.
- Problém přístupu na klasický pevný disk:
  - Čtení/zápis pracuje **s celými bloky** dat (např. sektor nebo cluster sektorů)
  - Operace čtení/zápisu se skládají z fáze vyhledání stopy (seek) a samotného načtení/zápisu bloku
  - **Seek má až několik milisekund, takže je potřeba jej dělat co nejméně často**
- Soubory indexů musíme optimalizovat tak, abychom při práci s nimi potřebovali co nejméně diskových seek-ů

Příklad rozložení diskových bloků

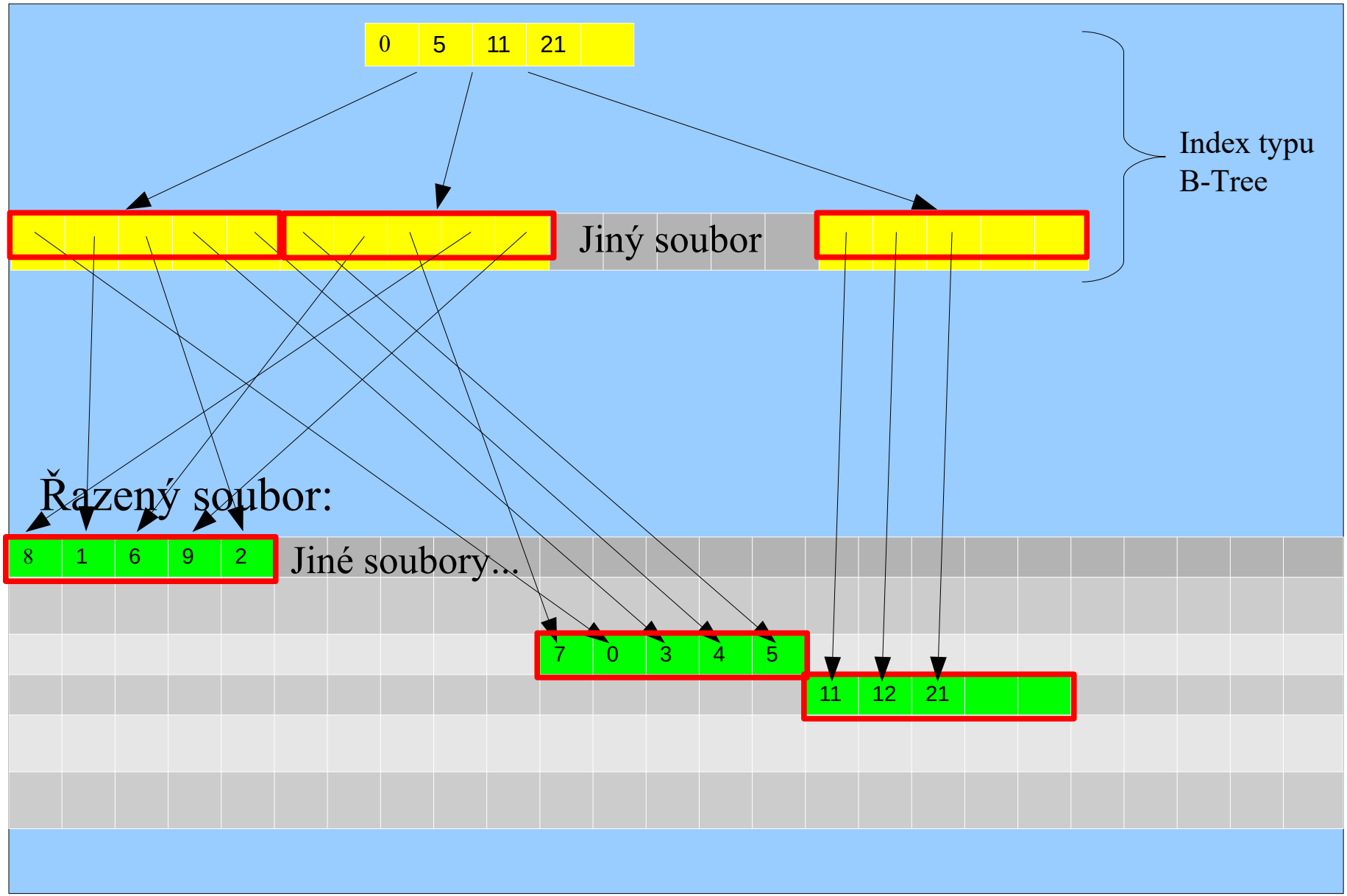


Optimalizace indexového souboru

- Indexový soubor na předchozím obrázku je uložen v nesouvislé sekvenci diskových bloků – pokud budeme hledat číslo 11, musí se udělat 1 seek navíc už v index-u.
- Řešení = index indexu :-)= B(+) Tree !



Disk:


0
5
11
21
Index typu
B-Tree
Jiný soubor
Řazený soubor:
8
1
6
9
2
Jiné soubory...
7
0
3
4
5
11
12
21

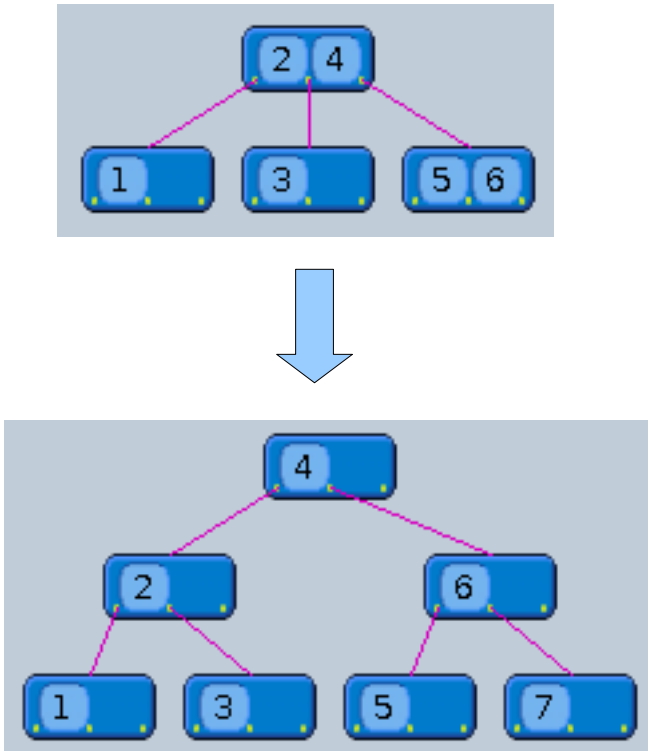
B-Tree

- Rudolf **B**ayer, Ed McCreight, **B**oeing Research Labs,1971
- Jedná se o stromy, kde vnitřní (ne-kořenové) uzly mohou mít *R* až *S* synů – tj. počet synů není fixní, ale variabilní v nějakém rozsahu.

  Každý vnitřní uzel tedy obsahuje *R*-1 až *S*-1 klíčů. Kořenový uzel pak 1 až *S*-1 klíčů.
- V ideálním případě kompletně zaplněných uzlů bude mít strom **výšku** = $\log_S(N+1)$, v nejhorším případě $\log_R((N+1)/2)$

  kde *N* je počet klíčů ve stromu.
- Většina uzlů není zcela zaplněna klíči, proto strom není potřeba tak často vyvažovat – nové klíče se většinou vlezou do stávajícího uzlu.
- Při vkládání do kompletně zaplněného uzlu je potřeba tento „rozdělit“ – tj. vytvořit 1 nový uzel a klíče původního uzlu rozdělit mezi tyto 2 uzly. Do rodiče pak vložit medián=hodnou oddělující tyto 2 uzly.
  Pokud je rodič zcela zaplněný, pokračuje algoritmus dělení i s ním.
- B strom je tímto udržován vyvážený.
- Kvůli snadnému dělení uzlů se volí $S=2*R$

Dělení uzlů při vkládání klíče



Více viz
http://slady.net/java/bt/view.php

B+ Tree

- Todo...

Odkazy na webu

- http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sorting/allSort.html - animace závodu řadících algoritmů
- http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html
- Nevíte, který algoritmus pro vaši aplikaci vybrat? http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE148.HTM