

IAL – 7. přednáška



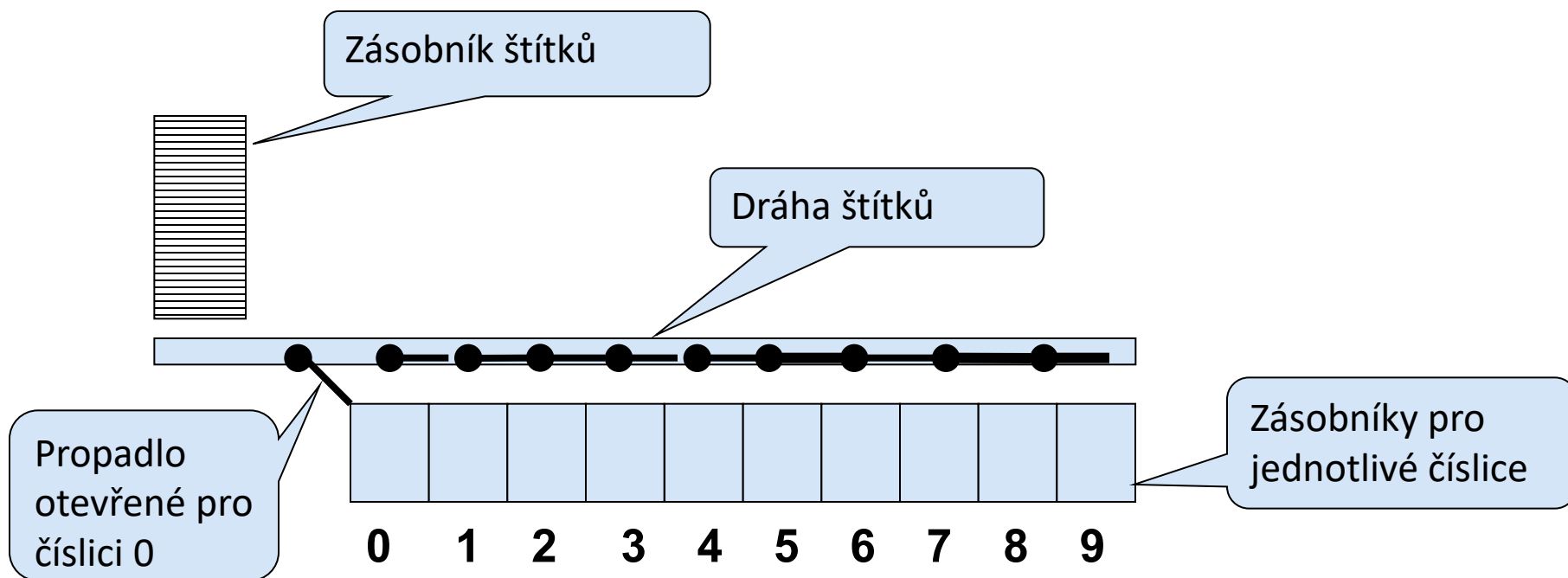
Řazení I.

Obsah přednášky

- Základní pojmy
 - Historie
 - Terminologie
 - Vlastnosti řadicích algoritmů
- Řazení podle více klíčů
- Řazení bez přesunu položek
- Klasifikace algoritmů řazení
- Řazení na principu výběru
 - Bubble sort a jeho varianty
 - Heap sort
- Řazení na principu vkládání
 - Bubble-insert sort
 - Binary-insert sort

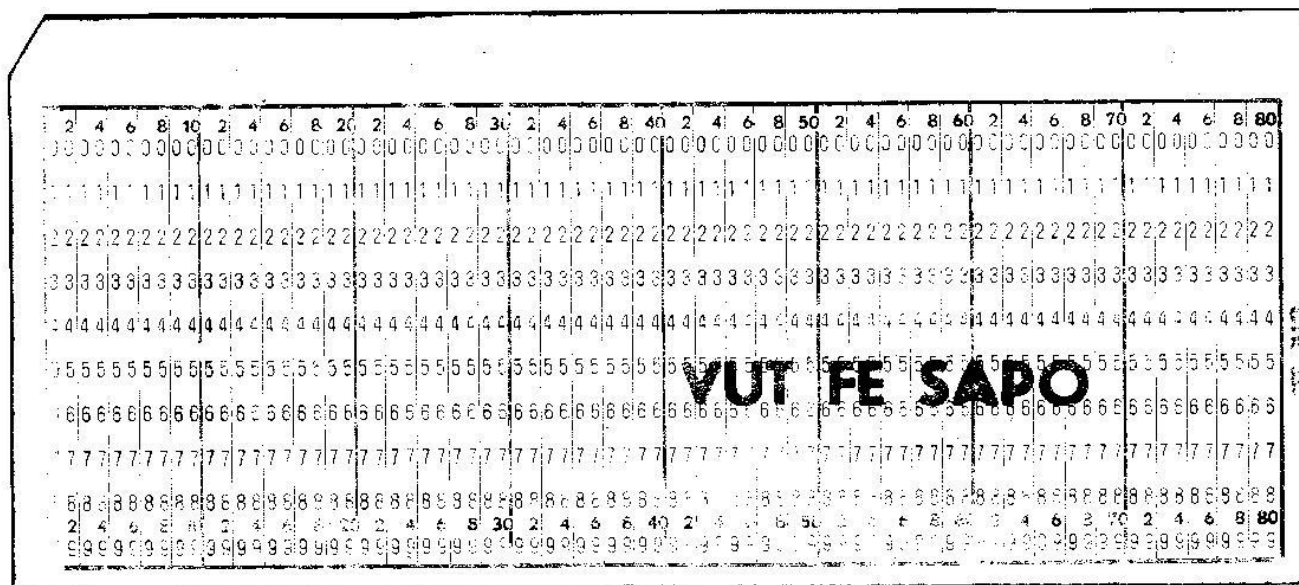
Řazení – historie

- Herman Hollerith použil *třídící* stroj pro sčítání obyvatelstva U.S.A. v r. 1890.



Řazení – historie

- ❑ Třídící stroj byl použit pro seřazení děrných štítků podle hodnoty čísla zapsaného pomocí dekadických číslic. Číslice byly reprezentované dírou na dané pozici v daném sloupci.
 - Štítek Hollerith měl 90 sloupců a obdélníkové dírky.
 - Štítek Aritma viz obr. měl 80 sloupců a kulaté dírky.Číslice 0 se neděrovala.



Řazení – historie

□ *Příklad řazení na třídícím stroji:*

- Seřazení štítků podle velikosti klíče, který byl reprezentován číslem vyděrovaným ve sloupcích 10, 11 a 12 proběhlo ve třech etapách (dáno počtem sloupců čísla).
- V první etapě se štítky třídícím strojem roztřídily do 10 skupin od 0 do 9 podle sloupce s nejnižší prioritou – tedy podle sloupce 12.
- Z 10 balíčků štítků se vytvořil jeden tak, že *nulový* balíček byl vespod, *jedničkový* byl nad ním ... a *devítkový* byl nahoře.
- Tento balík štítků se vložil do zásobníku a začala druhá etapa: třídění podle sloupce 11 se stejným postupem.
- Na konci poslední etapy byl získán balík seřazených štítků.

□ *Řazení bylo provedeno tříděním.*

Řazení tříděním – příklad

- Předpokládejme, že chceme seřadit následující množinu klíčů:
 $\{342, 835, 942, 178, 256, 493, 884, 635, 728\}$
Řazení provedeme opakovaným tříděním:

- Podle vzrůstající priority

342	728	178
942	835	256
493	635	342
884	342	493
835	942	635
635	256	728
256	178	835
178	884	884
728	493	942



- Podle klesající priority

178	728	342
256	635	942
342	835	493
493	342	884
635	942	635
728	256	835
835	178	256
884	884	728
942	493	178



Terminologie

- **Třídění** (angl. **sorting**) položek neuspořádané množiny je uspořádání do tříd podle hodnoty daného atributu – klíče položky.
Pozn.: Mezi třídami nemusí být definovaná relace uspořádání! (Třídíme směs jablek, hrušek a švestek do tří tříd.)
- **Řazení** (**ordering, sequencing**) je uspořádání položek podle relace lineárního uspořádání nad klíči.
 - **Dohoda:** Nebude-li explicitně stanoveno jinak, budeme předpokládat seřazení **od nejmenšího k největšímu**.



- *Pozn.:* Protože Hollerith dosahoval **řazení pomocí třídění na třídícím stroji**, používá se v praxi pro řazení v češtině i v angličtině nepřesné terminologie, **třídění** (*sorting*).

Terminologie

- **Setřídění** (**merging**) je vytváření souboru seřazených položek sjednocením několika souborů položek téhož typu, které jsou již seřazené.

Vlastnosti řadicích algoritmů

- **Přirozenost** – algoritmus se **chová přirozeně** pokud:
 - je doba potřebná k seřazení náhodně uspořádaného pole větší, než k seřazení již uspořádaného pole
 - a doba potřebná k seřazení opačně seřazeného pole je větší, než doba k seřazení náhodně uspořádaného pole.
 - Jinak říkáme, že se algoritmus **nechová přirozeně**.
- **Stabilita** vyjadřuje, zda mechanismus algoritmu zachovává relativní pořadí klíčů se stejnou hodnotou.
 - **Příklad: nestabilní** algoritmus může uspořádat sekvenci: 7, **5'**, 3, 1, **5''**, 9, 2, **5'''**, 8, 4, 6 s výsledkem: 1, 2, 3, 4, **5''**, **5'**, **5'''**, 6, 7, 8, 9.
 - **Stabilní** algoritmus vytvoří: 1, 2, 3, 4, **5'**, **5''**, **5'''**, 6, 7, 8, 9.

Řazení podle více klíčů

- V praxi je **řazení podle více klíčů** velmi časté.

Jako příklad mohou sloužit:

- *Řazení podle data narození*, kde datum sestává ze tří číselných klíčů: rok, měsíc a den.
- *Řazení studentů podle čtyř klíčů*: studijní program, ročník, studijní průměr a jméno. Úkolem je např. vytvořit seznam po programech, v programu po ročnících, v ročníku podle studijního průměru a studenty se stejným průměrem seřadit abecedně podle jména.

- Problém lze řešit třemi způsoby:

- Složená relace uspořádání
- Opakované řazení
- Aglomerovaný klíč

Řazení podle více klíčů

□ Vytvoření složené relace uspořádání:

```
bool function FirstOlder (TDateBorn first, second)
// vrací false, je-li druhý starší, nebo jsou oba stejně staří
    if first.year ≠ second.year
        return (first.year < second.year)
    else
        if first.month ≠ second.month // rok je shodný
            return (first.month < second.month)
        else // měsíc také shodný
            return (first.day < second.day)
```

Řazení podle více klíčů

□ Opakované řazení:

- Neuspořádanou množinu položek lze řadit postupně (opakovaně) podle **vzrůstající priority jednotlivých klíčů**. Podmínkou je použití **stabilní řadicí metody**!
- *Příklad:* Skupinu osob lze seřadit podle stáří tak, že se:
 1. Nejprve seřadí podle **dne** data narození
 2. pak se seřadí podle **měsíce** data narození
 3. a na konec se seřadí podle **roku** data narození.
- Tento způsob se podobá řazení děrných štítků v Hollerithově metodě.

Řazení podle více klíčů

□ Aglomerovaný klíč:

- Uspořádaná **N-tice klíčů se konvertuje na vhodný typ**, nad nímž je definována relace uspořádání.
- Vhodným typem může být např. typ řetězec.
- *Příklad aglomerovaného klíče*: Rodné číslo
(Lze pro řazení použít bez úpravy jako řetězec jen pro stejné pohlaví. Má tvar: RRMDDXXXX, ale ženy mají MM zvýšené o 50.)

K procvičení

- Napište funkci, která ze zadaného pole osob vytvoří seřazený seznam podle narozenin v roce. Při shodném datu narozenin má starší přednost.

```
typedef struct tdateborn
{
    int year, month, day;
}TDateBorn;
```

```
typedef struct tperson
{
    char *name;
    TDateBorn dateBorn;
}TPerson;
```

K procvičení

- Napište funkci libovolného algoritmu řazení pole, který znáte z prvního ročníku tak, aby se při volání funkce jedním vhodným parametrem ovládala složka, která bude klíčem řazení. Necht' **array** je pole prvků typu **TPerson**. Pak funkce:

void Sort (TArray array, TXX XX)

bude řadit jednou podle složky **year**, jindy podle složky **month** a jindy podle složky **day**, v závislosti na parametru **XX**. Nalezněte pro tento účel vhodný typ a deklarujte ho. Trojí volání této procedury pokaždé podle jiné složky může vytvořit seznam podle stáří nebo seznam podle narozenin.

K procvičení

- ❑ Napište funkci
bool FirstOlder (**char** *RC1, **char** *RC2)
- ❑ Napište funkci
bool EarlierBirthday (**char** *RC1, **char** *RC2)
- ❑ kde RC1 a RC2 jsou rodná čísla. U osob narozených ve stejný den má starší přednost před mladší, u stejně starých osob pak má přednost žena před mužem, a při stejném pohlaví rozhoduje pořadové číslo rodného čísla XXXX.

K procvičení

- Jsou dány typy:

```
typedef enum {bc,ing,phd}TProgram;
```

```
typedef struct tstudent  
{  
    char *name;  
    TProgram program;  
    int year;  
    float study_average;  
}TStudent;
```

- Vytvořte aglomerovaný (integrovaný) klíč pro vytvoření seznamů:
 - Podle studijního programu, v programu podle ročníku, v ročníku podle průměru, se stejným průměrem podle jména.
 - Podle průměru, se stejným průměrem podle programu, ve studijním programu podle ročníku, v ročníku podle jména.
- **Nápověda:** Aglomerovaný klíč bude typu řetězec. Průměr můžete převést na celé číslo např.: 2.75 $\Rightarrow$ 275. Řetězec můžete omezit např. na 20 znaků.

Řazení polí bez přesunu položek

- ❑ Nejčastěji prováděnými operacemi v algoritmech řazení jsou přesuny položek v poli a porovnávací operace.
- ❑ V případě *dlouhých* položek jsou přesuny časově velmi náročné
⇒ řazení polí **bez přesunu položek**.
- ❑ Implementace:
 - K řazenému poli vytvoříme **pomocné pole** (tzv. pořadník, *location*).
 - Po dokončení řazení pořadník udává, v jakém pořadí by měly být seřazeny položky původního pole (na první pozici pořadníku je index prvního prvku seřazeného pole atd.).
- ❑ Chceme-li mít na konci **seřazené pole**:
 - Přeskládáme prvky do výstupního pole s využitím pořadníku.
 - Prvky zřetězíme a přeskládáme do výstupního pole, nebo přeskládáme v poli samotném.

Řazení polí bez přesunu položek

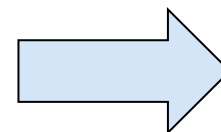
- Pořadník se inicializuje se hodnotami shodnými s indexem.

Ind Data Key

0		13
1		8
2		20
3		15
4		11

Location

0
1
2
3
4



Location

1
4
0
3
2

Řazení polí bez p. p. – implementace

□ Datové typy:

```
typedef struct telement{  
    TData data;  
    TKey key;  
}TElement  
  
#define MAX ...  
typedef TElement TArray[MAX];  
typedef int TLocation[MAX];  
  
    TArray array;  
    TLocation location;
```

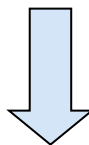
□ Inicializace:

```
for i ← (0, MAX-1):  
    location[i] ← i
```

Řazení polí bez přesunu položek

- Každé porovnání mezi dvěma prvky pole v normálním algoritmu řazení se v odpovídajícím algoritmu pro řazení bez přesunu položek transformuje tímto způsobem:

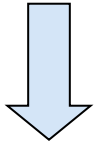
`array[i].key > array[j].key`



`array[location[i]].key > array[location[j]].key`

Řazení polí bez přesunu položek

- Každá výměna dvou prvků na indexech **i** a **j** v algoritmu řazení s přesunem se v zápisu algoritmu řazení bez přesunu transformuje takto:

`array[i] ↔ array[j]`

`location[i] ↔ location[j]`

- *Pozn.:* operace `↔` reprezentuje výměnu.

Řazení polí bez přesunu položek

- Pole seřazené bez výměny položek lze průchodem vložit do výstupního seřazeného pole `outArray` cyklem:

```
for i ← (0, MAX-1):  
    outArray[i] ← array[location[i]]
```

- Pole seřazené pomocí *pořadníku* lze také zřetěžit (prostřednictvím ukazatelů – indexů na další položku) a vytvořit seřazený seznam.

Zřetězení prvků

- Zřetězení prvků pole seřazeného bez přesunu položek:

□ first

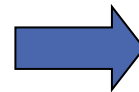
first 1

ind data key ptr

0		13	
1		8	
2		20	
3		15	
4		11	

location

1
4
0
3
2



ind data key ptr

0		13	3
1		8	4
2		20	-1
3		15	2
4		11	0

Zřetězení prvků

- Zřetězení lze provést následujícím kódem:

```
first ← location[0]
for i ← (0, MAX-2):
    array[location[i]].ptr ← location[i+1]
array[location[MAX-1]].ptr ← -1
                                // -1 slouží jako NULL
```

- Seřazenou zřetězenou posloupnost lze opět převést pomocí cyklu do seřazeného cílového pole nebo lze pomocí ukazatelů prvky přeskládat přímo v původním poli.

Klasifikace algoritmů řazení

□ Podle **přístupu k paměti**:

- metody vnitřního řazení (**řazení polí**) – přímý (náhodný) přístup
- metody vnějšího řazení (**řazení souborů a seznamů**) – sekvenční přístup

□ Podle **typu procesoru**:

- **sériové** (jeden procesor) – jedna operace v daném okamžiku
- **paralelní** (více procesorů) – více souběžných operací

Klasifikace algoritmů řazení

□ Podle principu řazení:

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

Smluvené konvence

- Metody řazení polí budeme vysvětlovat na zjednodušené struktuře s jednosložkovými položkami představovanými klíčem typu **int**:

```
typedef int TArray[MAX];
```

```
...
```

```
TArray A;
```

```
...
```

- Toto pole bude vstup/výstupním parametrem funkce řazení nebo globálním objektem.

Řazení na principu výběru (Select sort)

- ❑ Jádrem metody je **nalezení extrémního prvku** v zadaném segmentu pole a jeho výměna na konec (začátek) seřazené části pole.
- ❑ Takto je nalezeno $MAX-1$ minim (maxim), která jsou umístěna na svoji pozici.
- ❑ Princip metody:

...

for $i \leftarrow (0, MAX-2)$:

... *// Najdi nejmenší prvek mezi indexy **i** a **MAX-1**.*

... *// Jeho index ulož do pomocné proměnné **indexMin**.*

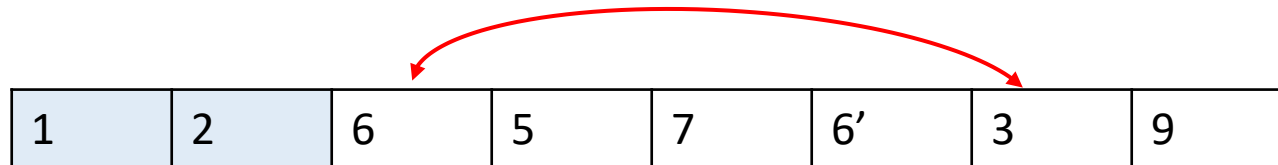
$A[i] \leftrightarrow A[indexMin]$

Select sort

```
procedure SelectSort (TArray A)
  for i ← (0, MAX-2):
    indexMin ← i // Poloha pomocného minima
    min ← A[i]   // Pomocné minimum
    for j ← (i+1, MAX-1):
      if min > A[j]:
        min ← A[j]
        indexMin ← j
    A[i] ↔ A[indexMin]
```

Select sort – zhodnocení

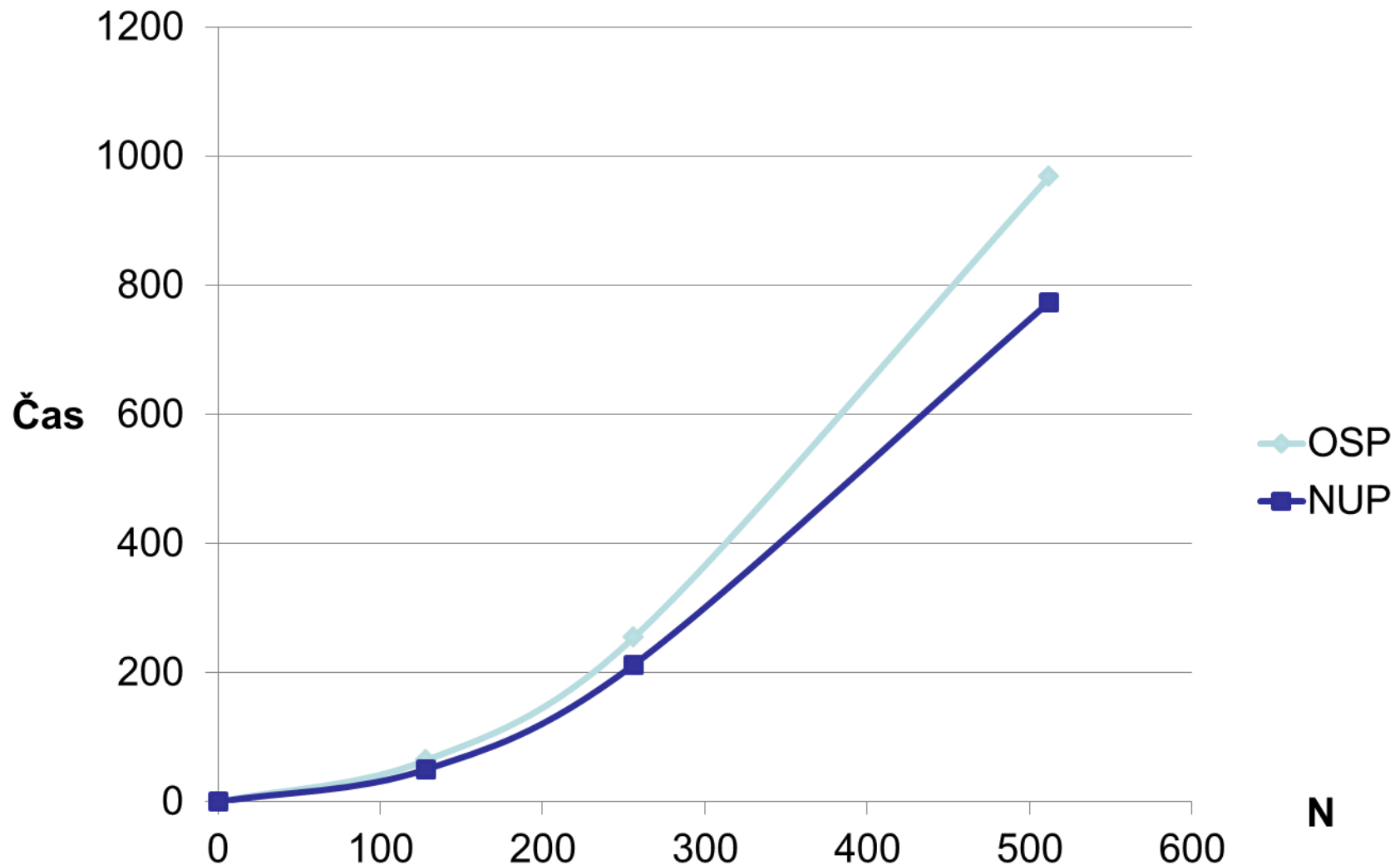
- Metoda je **nestabilní**. Vyměněný první prvek se může dostat za prvek se shodnou hodnotou.



- Má **kvadratickou časovou složitost**.
- Experimentálně byly naměřeny výsledky:
 - N je počet prvků
 - OSP je opačně seřazené pole
 - NUP je náhodně uspořádané pole

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

Select sort – naměřené výsledky graficky



Metoda *bublinového výběru* – Bubble sort

- Princip stejný jako u metody Select sort.
- Liší se **metodou nalezení extrému** a jeho přesunu:
 - 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.

Bubble sort – varianta zprava

```
procedure BubbleSort (TArray A)
  // průchod zprava - minimum doleva
  i ← 1
  do:
    finish ← true
    for j ← (MAX-1, i)-1:           // bublinový cyklus
      if A[j-1] > A[j]:
        A[j-1] ↔ A[j]
        finish ← false
    i ← i+1
  while (not finish) and (i < MAX)
```

Bubble sort – varianta zleva

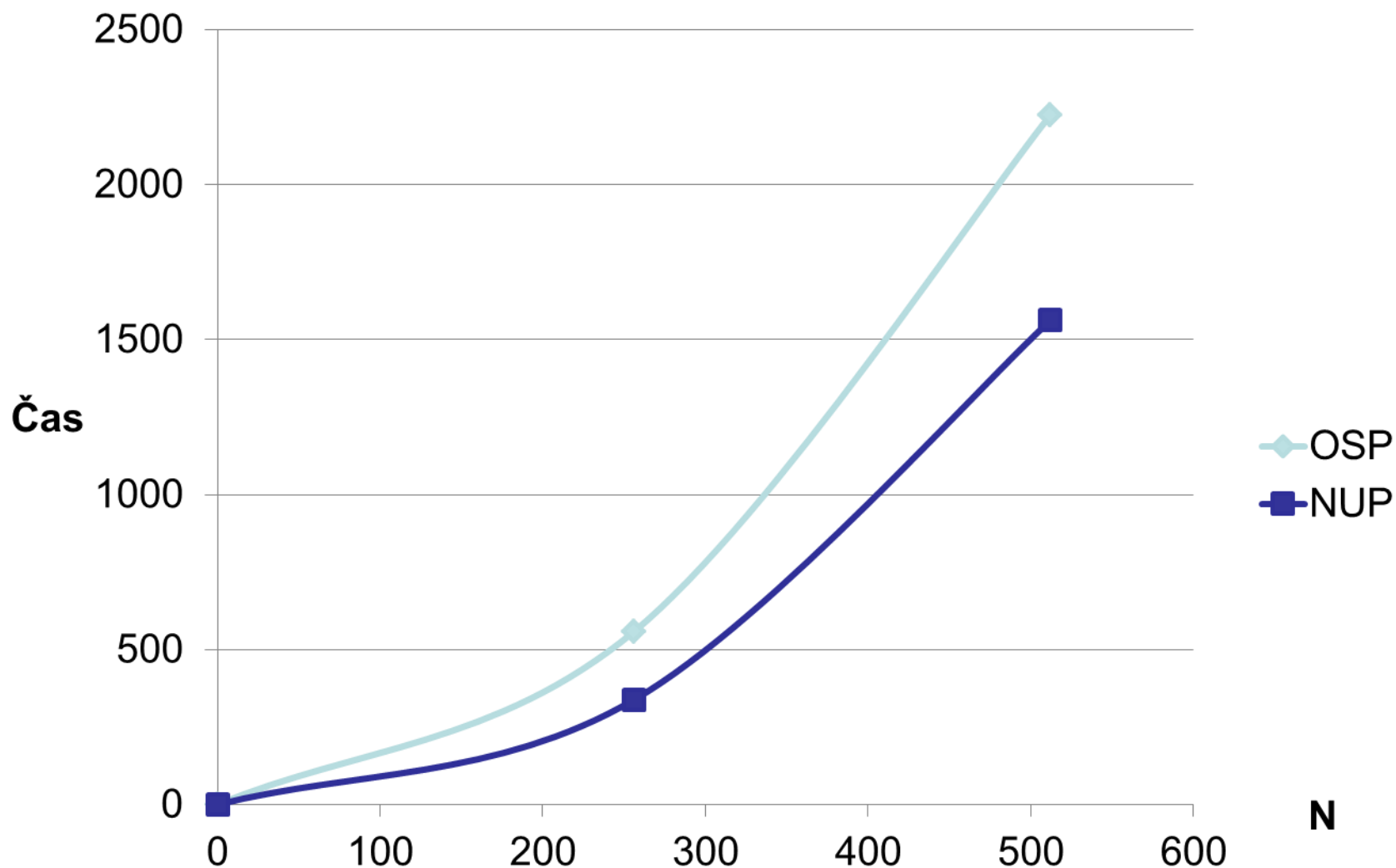
```
procedure BubbleSort2 (TArray A)
  // průchod zleva - maximum doprava
  auxN ← MAX-1
  continue ← true
  while continue and (auxN > 0):
    continue ← false
    for i ← (0, auxN-1):           // bublinový cyklus
      if A[i+1] < A[i]:
        A[i+1] ↔ A[i]
        continue ← true           // výměna - nelze skončit
  auxN ← auxN-1
```

Bubble sort – zhodnocení

- ❑ Bublinový 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 nejrychlejší metoda v případě, že pole je již seřazené!
- ❑ Experimentálně naměřené hodnoty:

n	256	512
NUP	338	1562
OSP	558	2224

Bubble sort – naměřené výsledky graficky



Bubble sort – varianty

- Od Bubble sortu byla odvozena řada vylepšených variant:
 - **Ripple sort:** pamatuje si 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 *na své místo* a teprve pak pokračuje dál. Končí tím, že nevymění nejpravější dvojici.

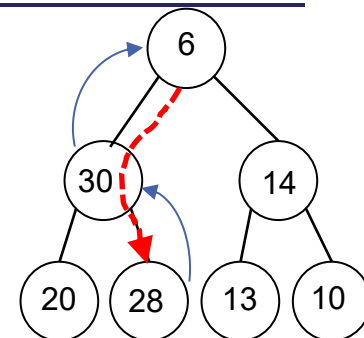
- **Pozn.:** Varianty bohužel nemají významnější efekt z pohledu časové složitosti algoritmů. Používají ale zajímavé programátorské techniky.

Řazení hromadou – Heap sort

- **Hromada (*halda, heap*)** 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 je **binární hromada**, která je založená na binárním stromu, pro který navíc platí:
 - Všechny hladiny kromě poslední jsou plně obsazené.
 - Poslední hladina je zaplněna zleva.

Rekonstrukce hromady

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



Heap sort – princip

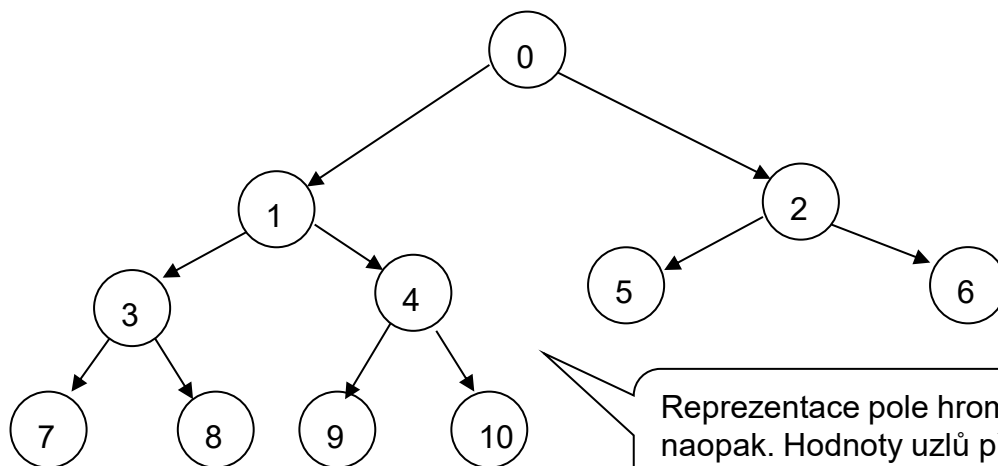
- ❑ Hromada má v **kořeni** vždy **maximální nebo minimální** prvek.
- ❑ Prvek z kořene představuje **extrémní prvek** v dané množině a může být *vložen* na své místo do výstupního pole.
- ❑ Jakou hodnotou můžeme **nahradit prvek v kořeni** – hodnotou nejnižšího a nejpravějšího uzlu.
- ❑ Hromada pak bude porušena v kořeni – to ale umíme napravit **zatřesením** v čase $\log_2 n$.
- ❑ Provedeme-li odebrání prvku a **zatřesení** hromadou pro všechny prvky, získáme seřazenou posloupnost s lineární složitostí $n * \log_2 n$.
- ❑ Problém – jak vytvořit heap?

Implementace hromady polem

□ Hromadu lze implementovat polem:

- Protože musí být zaplněny všechny hladiny kromě poslední a poslední musí být zaplněna zleva, můžeme strom *ukládat* do pole po hladinách.
- Pak **platí pro otcovský a synovské uzly vztah**:
když je otcovský uzel na indexu i , pak je levý syn na indexu $2i+1$ a pravý syn na indexu $2i+2$.

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----



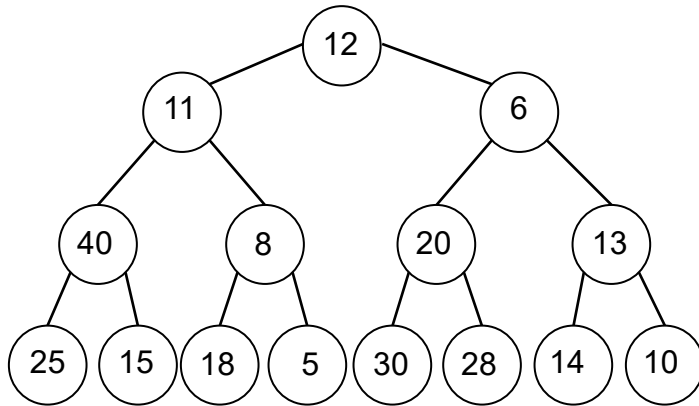
Reprezentace pole hromadou a naopak. Hodnoty uzlů představují indexy pole.

Vytvoření hromady

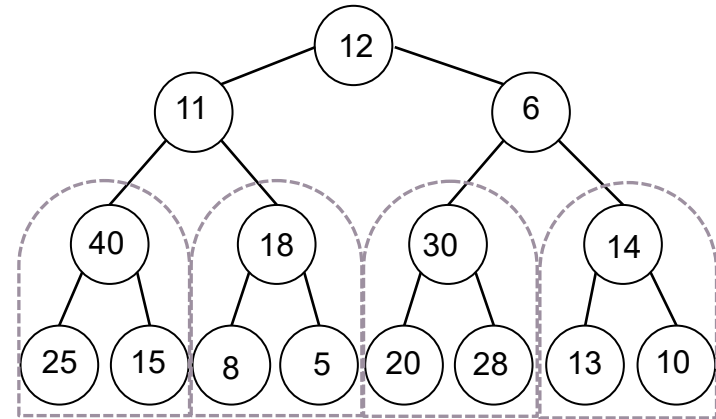
- Při využití operace *Sift* lze hromadu ustavit takto:
 - Vždy potřebujeme hromadu, která je porušena pouze v kořeni.
 - Začneme s *nejnižším a nejpravějším otcovským uzlem* – ten je kořenem hromady (podstromu), která je porušena v kořeni. Operací *Sift* opravíme.
 - Dále *postupujeme po všech otcovských uzlech doleva a nahoru* až k hlavnímu kořeni.
- Jak najdeme potřebné otcovské uzly?
 - Má-li pole `MAX` prvků (indexováno od 0 do `MAX-1`), pak nejnižší a nejpravější otcovský uzel odpovídající hromady má index:
 $(MAX \div 2) - 1$
 - Následující otcovské uzly leží na předchozích indexech.
 - Celkem musíme opravit $n/2$ hromad, celé ustavení hromady zvládneme v čase $n/2 * \log_2 n$.

Vytvoření hromady

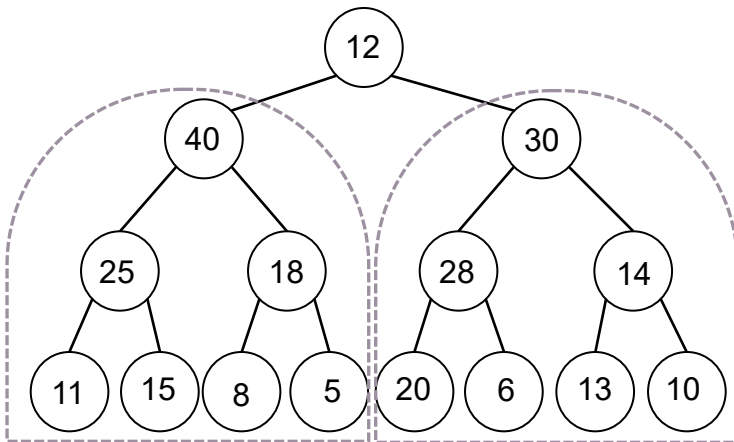
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	11	6	40	8	20	13	25	15	18	5	30	28	14	10



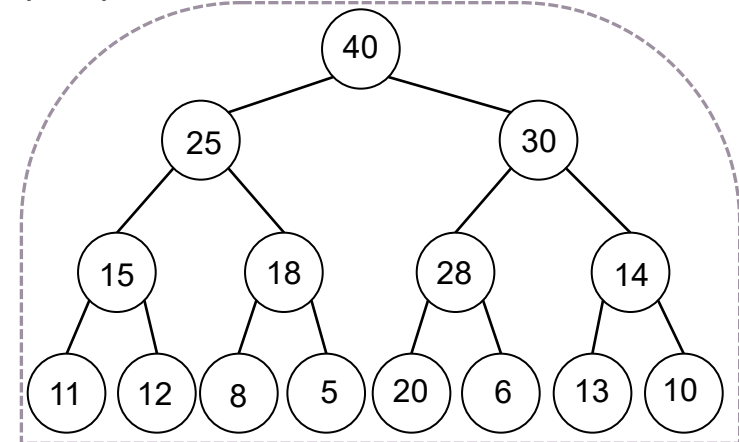
Neuspořádané pole



Vytvoření 4 hromad (zprava doleva) na předposlední úrovni



Vytvoření 2 hromad (zprava doleva) na 2. úrovni



V posledním kroku lze ze 2 hromad vytvořit jedinou hromadu

Vytvoření hromady

```
// ustavení hromady
left ← (MAX div 2) - 1 // nejnižší a nejpravější otec
right ← MAX - 1
for i ← (left, 0)-1:
    SiftDown(A, i, right)
```

Pozn.: Parametry metody `SiftDown`: pole s prvky, index kořene a index nejnižšího nejpravějšího syna.

Heap sort

```
procedure HeapSort (TArray A)
  // ustavení hromady
  left ← (MAX div 2) - 1      // nejnižší a nejpravější otec
  right ← MAX - 1
  for i ← (left, 0)-1:
    SiftDown(A, i, right)

  // vlastní cyklus Heap-sortu
  for right ← (MAX - 1, 1)-1:
    A[0] ↔ A[right]
    // výměna kořene s akt. posledním prvkem
    SiftDown(A, 0, right - 1) // znovustavení hromady
```

Prosetí prvku

- Při implementaci binárního stromu polem je důležité poznat konec větve (terminální uzel nebo uzel, který nemá pravého syna):
 - Je-li $2^{*i+1} > \text{Max}-1$, je uzel i terminální.
 - Je-li $2^{*i+1} = \text{Max}-1$, má uzel i jen levého syna.
 - Je-li $2^{*i+1} < \text{Max}-1$, má uzel i oba syny.

```

procedure SiftDown (TArray A, int left, int right)
// left je index kořenového uzlu, který porušuje heap,
// right je index posledního prvku heapu

i ← left
j ← 2*i+1           // index levého syna
temp ← A[i]         // pomocná proměnná
continue ← j ≤ right // řídicí proměnná cyklu
while continue:
    if j < right:           // uzel má oba syny
        if A[j] < A[j+1]    // pravý syn je větší
            j ← j+1         // pokračujeme tedy s ním
    if temp ≥ A[j]: // temp našel své místo = konec
        continue ← false
    else: // temp padá níž, A[j] jde o úroveň výš
        A[i] ← A[j]
        i ← j           // syn je otcem v dalším cyklu
        j ← 2*i+1       // nový levý syn
        continue ← j ≤ right // pokračujeme až na list
A[i] ← temp // konečná pozice „propadajícího“ kořene

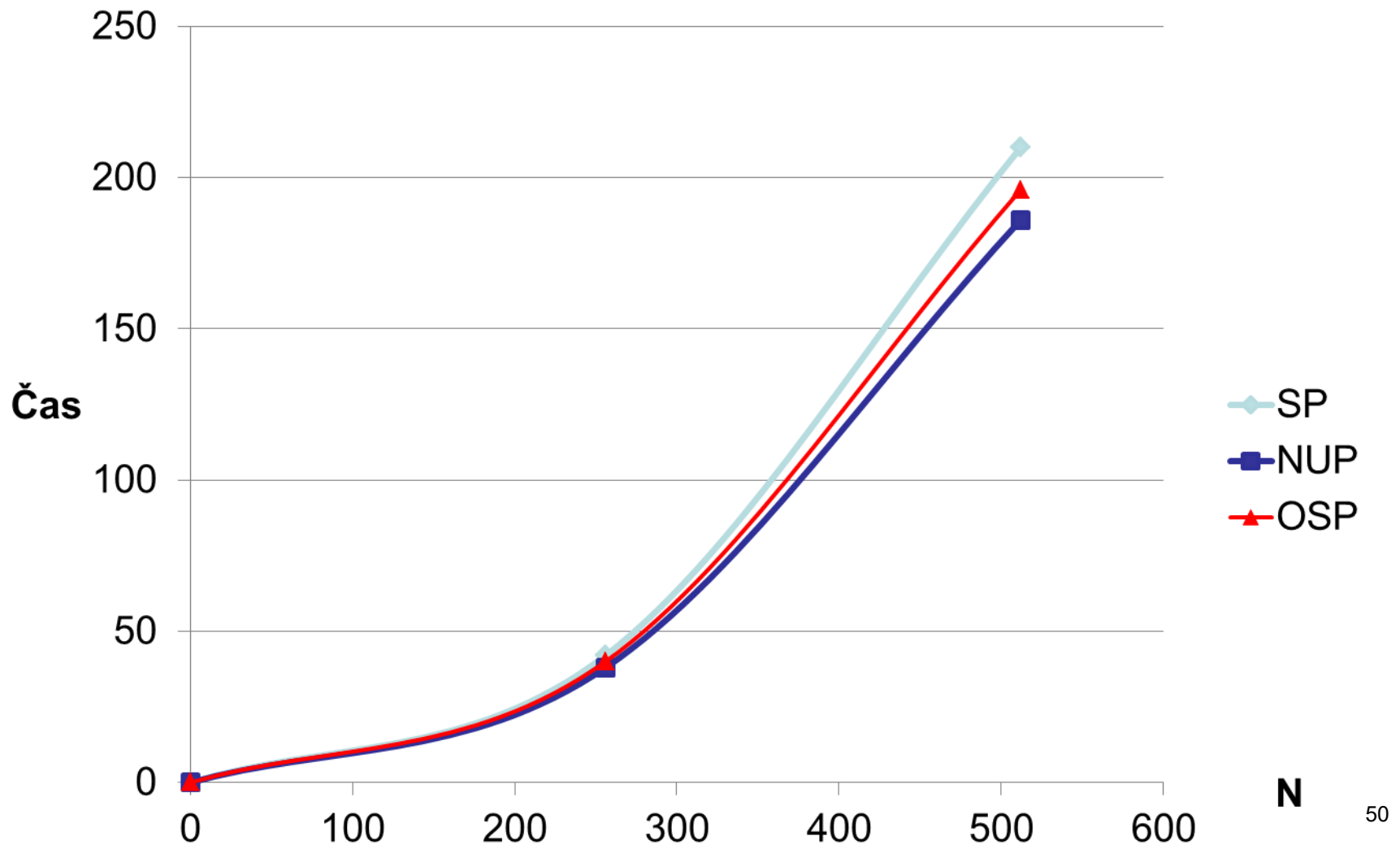
```

Heap sort – zhodnocení

- Heap sort je řadící metoda s **lineárním** složitostí, protože **sift** umí rekonstruovat hromadu (najít extrém mezi N prvky) s logaritmickou složitostí.
- Heap sort je **nestabilní** a nechová se přirozeně.
- Naměřené hodnoty:

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

Heap sort – naměřené výsledky graficky



K procvičení

- Je dána hromada o N prvcích typu `int` v poli `H`.
Napište funkci, která rekonstruuje (znovuustaví)
hromadu porušenou zápisem libovolné hodnoty na
index $0 \leq K < N$.

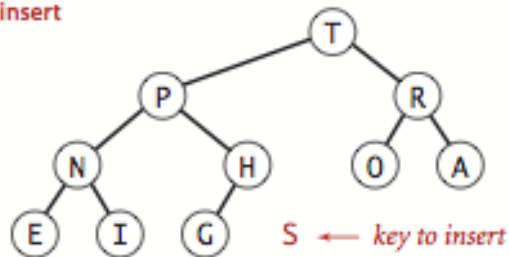
Další využití hromady

- **Prioritní frontu** lze implementovat binární hromadou:
 - Prvky jsou organizovány na základě jejich priority
 - V kořeni bude vždy prvek s maximální/minimální prioritou.
 - Operace **Front** (nalezení maxima/minima) odpovídá čtení hodnoty z kořene: **$O(1)$**
 - Operace **Remove** – prvek z kořene musí být odebrán a nahrazen jiným prvkem. Hromada bude porušena a je potřeba ji napravit: **$O(\log n)$**
 - Operace **Insert** – prvek je vložen na nejnižší hladinu na první volné místo a musí vybublat nahoru: **$O(\log n)$**

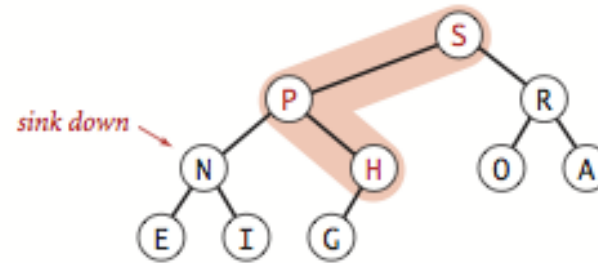
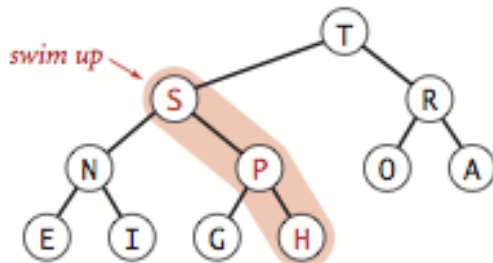
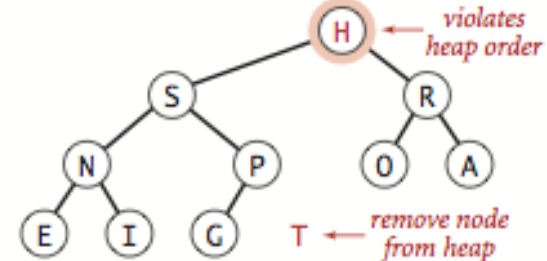
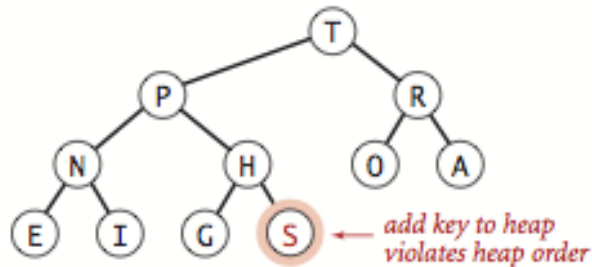
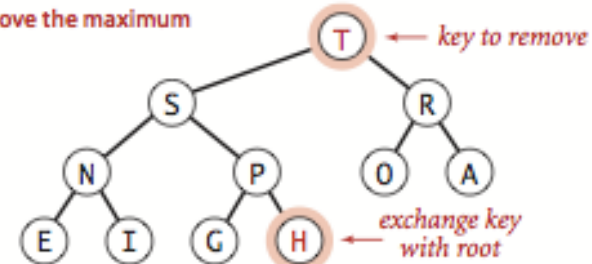
- **Pozn.:** Pozor, takto implementovaná prioritní fronta neumí bez dalšího rozšíření zachovat vzájemné pořadí prvků se stejnou prioritou.

Prioritní fronta

insert



remove the maximum



Heap operations

Prioritní fronta

- Možné způsoby implementace prioritní fronty:
 - Implementace nesetříděným polem nebo spojovým seznamem:
 - Vložení prvku: $O(1)$
 - Odebrání/nalezení prvku s nejvyšší prioritou: $O(n)$
 - Implementace setříděným polem nebo seznamem:
 - Vložení prvku: $O(n)$
 - Odebrání/nalezení prvku s nejvyšší prioritou: $O(1)$
 - Implementace (binární) haldou:
 - Vložení prvku: $O(\log n)$
 - Odebrání libovolného prvku s nejvyšší prioritou: $O(\log n)$
 - Nalezení prvku s nejvyšší prioritou: $O(1)$

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.
- ❑ Pole dělíme na dvě části:
 - Levou – seřazenou a pravou – neseřazenou.
 - Levou část tvoří na začátku první prvek.
- ❑ Algoritmus řazení má následující strukturu:

for $i \leftarrow (1, \text{MAX}-1)$:

```
// najdi v levé části index k, kam 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
```

Bubble-insert sort

- Metoda bublinového vkládání
- Kombinuje vyhledání místa pro vkládání i posun segmentu pole do jednoho cyklu:
 - Postupným porovnáváním a výměnou dvojic prvků.
- *Pozn.:* Tato metoda řazení odpovídá dříve zmíněné metodě *Shuttle sort*, která byla odvozena od *Bubble sortu* (tedy od metody založené na principu výběru). Postup řazení ale odpovídá spíše principu vkládání.

Bubble-insert sort

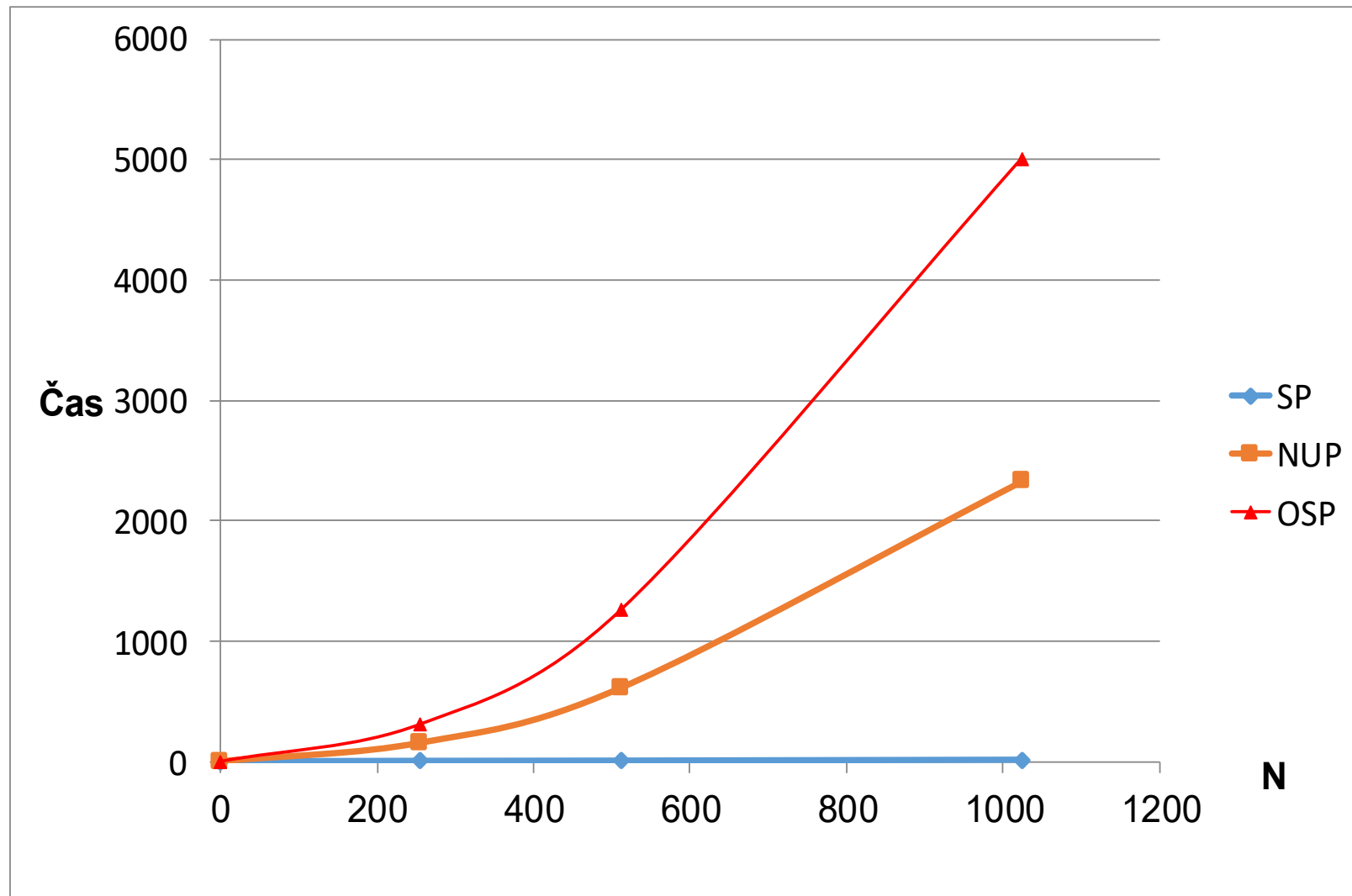
```
procedure BubbleInsertSort (TArray A)
  for i ← (1, MAX-1):
    tmp ← A[i]
    j ← i-1
    while j ≥ 0 and tmp < A[j]: // zkrat. vyhodnocování!
                                  // najdi a posuň prvek
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← tmp                // konečné vložení na místo
```

Bubble-insert sort – zhodnocení

- Metoda je **stabilní** – je vhodná pro vícenásobné řazení podle více klíčů.
- Chová se **přirozeně** a pracuje **in situ**.
- Má **kvadratickou** časovou složitost.
- Experimentálně byly naměřeny tyto hodnoty:

N	256	512	1 024
SP	4	6	14
NUP	156	614	2 330
OSP	312	1 262	5 008

Bubble-insert sort – naměřené výsledky graficky



Binary-insert sort

- ❑ Vkládání s binárním vyhledáváním.
- ❑ Pro vložení prvku vyhledáváme místo v **seřazené posloupnosti** – lze využít **binární vyhledávání**.
- ❑ V případě shodných klíčů musí metoda nalézt místo **za nejpravějším ze shodných klíčů** – varianta Dijkstrový metody binárního vyhledávání.

Binary-insert sort

```
procedure BinaryInsertSort (TArray A)
  for i  $\leftarrow$  (1, MAX-1):
    tmp  $\leftarrow$  A[i]
    left  $\leftarrow$  0                // hranice již seřazené části
    right  $\leftarrow$  i-1
    while left  $\leq$  right:        // binární vyhledání
      m  $\leftarrow$  (left+right) div 2
      if tmp < A[m]:
        right  $\leftarrow$  m-1
      else:
        left  $\leftarrow$  m+1        // ale skončíme těsně za
    for j  $\leftarrow$  (i-1, left)-1:
      A[j+1]  $\leftarrow$  A[j]      // posun segmentu pole doprava
    A[left]  $\leftarrow$  tmp        // prvek z pozice i na své místo
```

Binary-insert sort zhodnocení

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

N	256	512	1 024
NUP	134	502	1 024
OSP	248	956	3 736

Binary-insert sort – naměřené výsledky graficky

