



# Algoritmy pro vyhledávání a řazení

---

Jitka Kreslíková, Aleš Smrčka

2021

Fakulta informačních technologií  
Vysoké učení technické v Brně

IZP – Základy programování



# Algoritmy pro vyhledávání a řazení

---

- ☐ Metody vyhledávání
- ☐ Metody řazení



# Algoritmy pro vyhledávání

---

- ❑ Vyhledávání (searching)
  - Algoritmy pro co nejefektivnější nalezení hledané informace.
  - Ne vždy lze v datech vyhledávat efektivně.
  - Rychlé vyhledávání → nutno podniknout jisté kroky už při ukládání dat.
  - Někdy nazýváno Ukládání a vyhledávání dat (Data Storage and Retrieval)
- ❑ Proč vyhledáváme určité údaje?



# Klasifikace vyhledávacích algoritmů

---

## □ Podle místa uložení dat

- **Interní** – data jsou uložena v operační paměti
- **Externí** – data jsou uložena na disku nebo v jiné externí paměti
- **Kombinované** – nalezení bloku dat na disku a dohledání v rámci nalezeného bloku v operační paměti



# Klasifikace vyhledávacích algoritmů

---

## □ Podle principu vyhledávání

- **Sekvenční** – sekvenčně se prochází prohledávaná struktura,
- **Indexsekvenční** – přímým přístupem se nalezne blok a v něm se sekvenčně dohledá ([ISAM](#) - ***I**ndexed **S**equential **A**ccess **M**ethod*)

[on line, cit. 2020-11-12]

- **Vyhledávání v tabulkách s rozptýlenými položkami** (hash table) – index hledaného prvku se „vypočte“ speciální funkcí.
- **Nesekvenční v seřazeném poli** – index prvku se nalezne rychleji než sekvenčním průchodem,
- **V seřazených stromech** – speciálně organizované struktury pro uložení prohledávaných dat,



# Klasifikace vyhledávacích algoritmů

---

## □ Podle práce s klíči

- Klíč = část prvku, podle kterého se vyhledává
  - Index, příjmení, rodné číslo, SPZ, výška, ...
- **Původní** klíče
  - Vyhledává se přímo podle hodnoty klíče
- **Transformované** klíče
  - Původní klíč → úprava → transformovaný → vyšší efektivita vyhledávání.
  - Vedou k tabulkám s rozptýlenými položkami.



# Klasifikace vyhledávacích algoritmů

---

## □ Jednorozměrné vyhledávání

### ■ Adresní

- Využívá jednoznačný vztah mezi hodnotou klíče a jeho umístěním ve vyhledávacím prostoru  $S$ .
- Přímý přístup k prvkům pole pomocí indexů.

### ■ Asociativní

- Porovnávání prvků
- Při hledání prvku  $x \in S$  se využívají relace (porovnávání) mezi prvky prostoru  $S$
- Mezi prvky musí existovat relace uspořádání.
- Většinou požadujeme, aby byl prostor prvků uspořádán podle této relace.



# Klasifikace vyhledávacích algoritmů

---

## ❑ Vícerozměrné vyhledávání

- Zároveň podle více klíčů = **složený klíč**.
- Nutno přizpůsobit datové struktury i algoritmy.
- Dotazy lze rozdělit do tří kategorií
  - Dotazy na **úplnou shodu** – požaduje se shoda na všech klíčích.
  - Dotazy na **částečnou shodu** – požaduje se shoda jen na některých složkách klíče.
  - Dotazy na **intervalovou shodu** – požaduje se, aby hledaný klíč ležel v určeném intervalu.

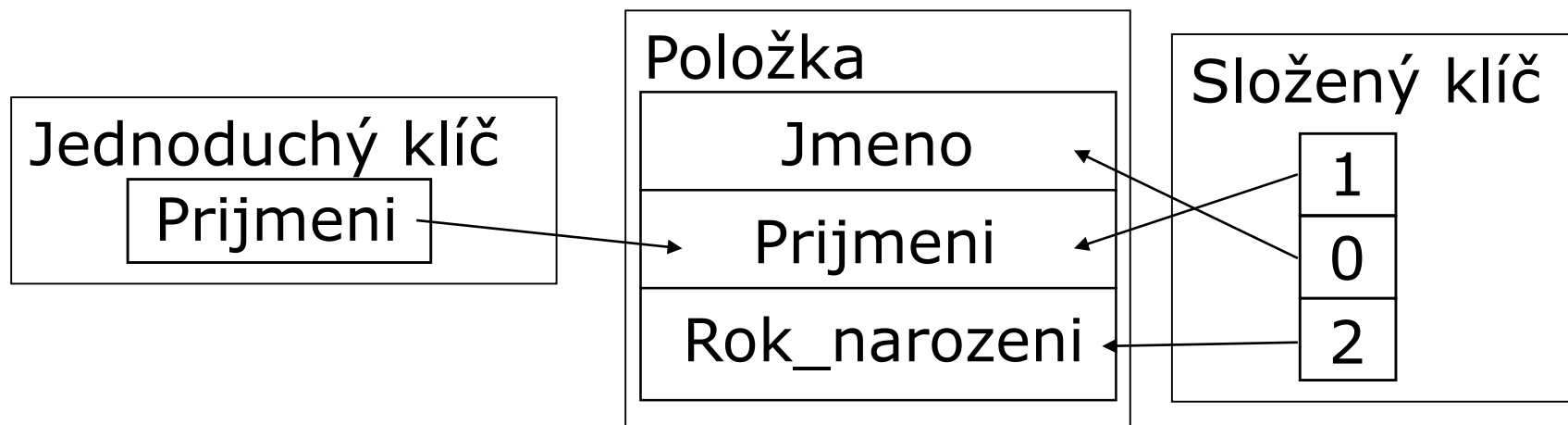




# Klasifikace vyhledávacích algoritmů

## □ Složený klíč

- Více jednoduchých klíčů
- Každá složka může obsahovat data různých datových typů
- Porovnávání = zřetězení porovnávacích operací nad jednotlivými složkami ve správném pořadí.





# Sekvenční vyhledávání

---

- ❑ Základní princip
  - Sekvenční průchod všemi položkami až do nalezení hledané položky nebo dosažení konce
- ❑ Datová struktura pro efektivní ukládání a vyhledávání informací → **tabulka**,
  - jednoduchá reprezentace → **pole**
  - prvky – struktury obsahující klíč a hodnotu,
- ❑ Pole
  - **Uspořádané** – podle hodnot klíčů
  - **Neuspořádané** – prvky jsou ukládány bez ohledu na hodnotu klíče



# Sekvenční vyhledávání

---

- ❑ Vyhledávání prvku se zadaným klíčem znamená postupné prohledání – **sekvenční zpracování**.
- ❑ Nechť je dán datový typ:

```
typedef struct tpolozka
{
    typData data;
    typKlice klic;
} TPolozka;
```

- ❑ Nad typem **typKlice** je definována relace ekvivalence.
  - ❑ Dále budeme používat funkce **isEqual** a **isLess** vracející hodnotu typu **bool**.
-



# Sekvenční vyhledávání

---

## □ Poznámka 1

- Porovnávání určitých typů dat není triviální operace, zvláště při použití složených klíčů.

## □ Poznámka 2

- Struktura `TPolozka` obsahuje složku `klic` pouze pro názornost.
- V praktických aplikacích většinou tvoří klíč samotné datové složky struktury.

## □ Další deklarace

```
TPolozka pole[N]; // tabulka  
typKlice k;       // hledaný klíč
```



# Sekvenční vyhledávání

*Příklad:* algoritmus sekvenčního vyhledávání v poli.

```
int i=0;
while (i<N && !isEqual(pole[i].klic, k))
    i++;
bool found = (i < N);
if (found) ... //prvek pole[i] je hledaným prvkem
```

**Úkol:** pokud by místo

`bool found = (i < N);` bylo použito:

`bool found = isEqual(pole[i].klic, k);`, mohlo by dojít k přístupu za hranici pole.

Pro který případ (jediný) by tato situace mohla nastat?



# Sekvenční vyhledávání

---

- ❑ Sekvenční vyhledávání v poli se zarážkou
  - jednoduchou úpravou lze tento algoritmus zrychlit zjednodušením podmínky pro konec cyklu,
  - pole vytvoříme o jeden prvek delší – na konec se vloží hledaný klíč – zarážka,
  - cyklus skončí vždy „úspěšným vyhledáním“, ke kterému dojde buď až na zarážce (neúspěšné vyhledání) nebo dříve (úspěšné vyhledání).



# Sekvenční vyhledávání

*Příklad:* sekvenční vyhledávání v poli se zarážkou.

```
TPolozka pole[N+1];  
...  
pole[N].klic = k; // vložení zarážky  
int i=0;  
while(!isEqual(pole[i].klic, k))  
    i++;  
bool found = (i != N);  
if (found) ... //prvek pole[i] je hledaným prvkem
```

Poznámka: při rušení položky z neseřazeného pole není nutné posouvat všechny prvky – stačí přesunout poslední prvek na místo rušeného prvku a snížit velikost pole.



# Sekvenční vyhledávání

*Příklad:* sekvenční vyhledávání v seřazeném poli.

```
int i=0;
while (i<N && isLess(pole[i].klic, k))
    i++;
bool found = (i<N) && isEqual(pole[i].klic, k);
if (found) ... //prvek pole[i] je hledaným prvkem
```

## ❑ Dynamické vlastnosti vyhledávání v seřazeném poli

- Při vkládání a rušení prvku nutno zachovat uspořádanost – může jít o „drahou“ operaci.
- Do určitého počtu nových prvků je lepší přidávat na konec a neuspořádanou část projít sekvenčně.





# Sekvenční vyhledávání

*Příklad:* vyhledávání v seřazeném poli se zarážkou.

```
TPolozka pole[N+1];  
...  
pole[N].klic = maxKlic;  
// prvek s maximální hodnotou (klíčem)  
// maxKlic je prvek s větší hodnotou (klíčem)  
// než hodnoty všech prohledávaných prvků (klíčů)  
int i=0;  
while (isLess(pole[i].klic, k))  
    i++;  
bool found = isEqual(pole[i].klic, k);  
if (found) ... //prvek pole[i] je hledaným prvkem
```



# Nesekvenční vyhledávání

---

## □ Binární vyhledávání

- Necht' pro pole implementující vyhledávací tabulku platí:

$\text{pole}[0].\text{klic} < \text{pole}[1].\text{klic} < \dots < \text{pole}[N-1].\text{klic},$

- dále necht' pro vyhledávaný klíč  $k$  platí:

$k \geq \text{pole}[0].\text{klic}$  a  $k \leq \text{pole}[N-1].\text{klic}$

- Poznámka

- Zde uváděné porovnávání – pouze zjednodušující abstrakce.



# Nesekvenční vyhledávání

---

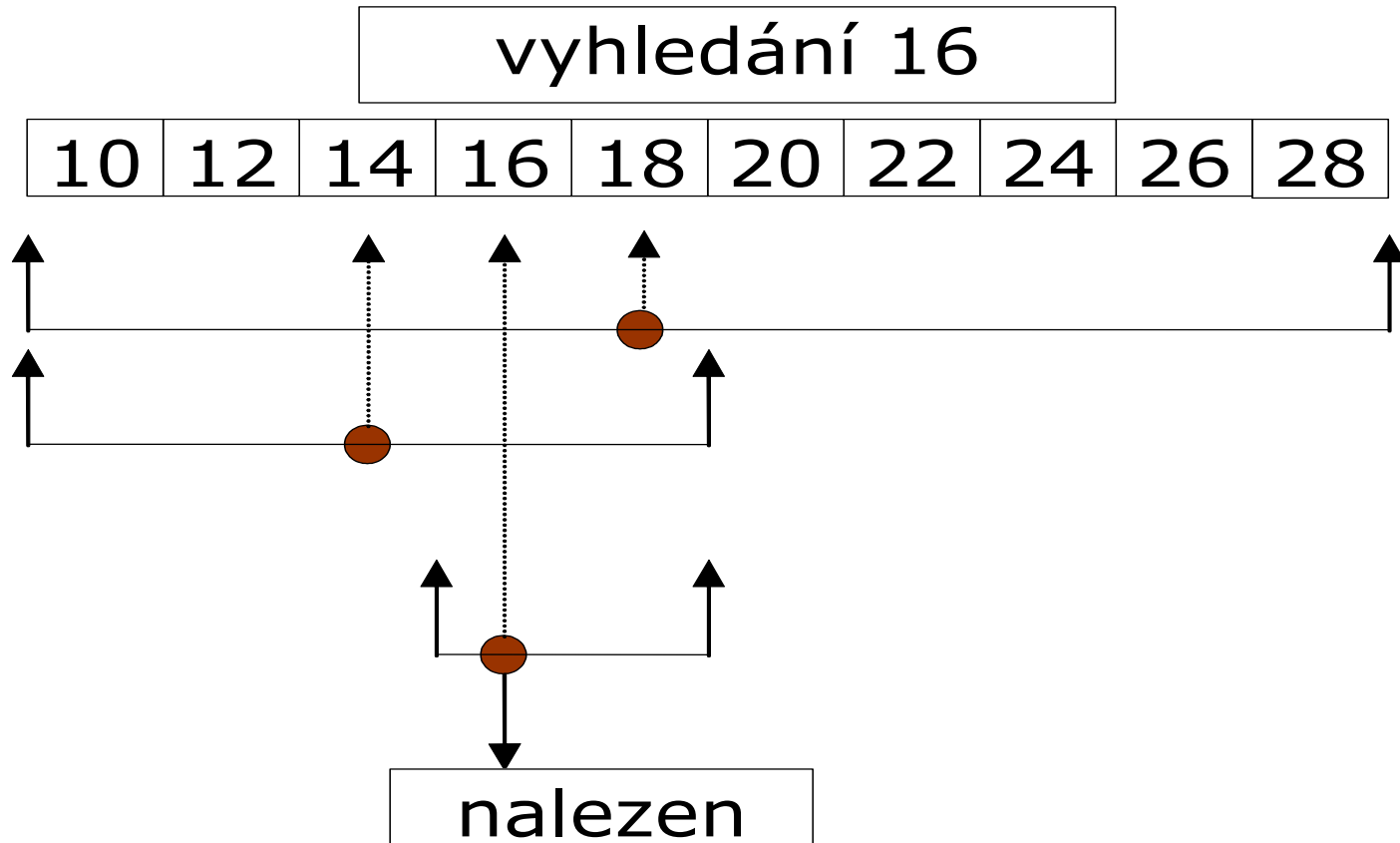
## **Princip binárního vyhledávání:**

- ❑ vyhledávaný klíč se porovná s klíčem položky, která je umístěna v polovině vyhledávaného pole,
- ❑ dojde-li ke shodě, končí vyhledávání úspěšně,
- ❑ je-li vyhledávaný klíč menší, postupuje se porovnáváním prostředního prvku v levé polovině původního pole,
- ❑ je-li větší v pravé polovině původního pole,
- ❑ vyhledávání končí neúspěchem v případě, že prohledávaná část pole je prázdná (její levý index je větší než pravý).



# Binární vyhledávání

*Příklad:* binární vyhledávání v seřazeném poli.





# Nesekvenční vyhledávání

*Příklad:* binární vyhledávání v seřazeném poli.

```
int search(Tpolozka a[], typKlice k, int l, int r)
{
    while (r >= l)
    {
        int m = (l + r) / 2;
        if (isEqual(k, a[m].klic)) return m;
        if (isLess(k, a[m].klic))
            r = m - 1;
        else
            l = m + 1;
    }
    return -1;    // nenalezen
}
```



# Nesequenční vyhledávání

---

## □ Binární vyhledávání

- Délka prohledávané posloupnosti se po každém porovnání zmenší na polovinu.
- Složitost metody:  $O(\log_2(N))$



# Algoritmy pro řazení

---

- ☐ Vlastnosti metod řazení.
- ☐ Problematika porovnání český řetězců.
- ☐ Řazení podle více klíčů.
- ☐ Řazení bez přesunu položek.
- ☐ Metoda přímého výběru.



# Obscná formulace problému řazení

---

- Je dána neprázdná množina

$$A = \{a_1, a_2, \dots, a_n\},$$

- je potřeba najít permutaci  $\pi$  těchto  $n$  prvků, která zobrazuje danou posloupnost do neklesající (nerostoucí) posloupnosti:

$$a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}$$

tak, že:  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)},$

- množinu  $U$ , ze které vybíráme prvky tříděné množiny, nazýváme **univerzum**,





# Obscná formulace problému řazení

---

- ❑ prvky množiny ***U*** se nazývají ***klíče*** a informace vázaná na klíč spolu s klíčem se nazývá ***záznam***
- ❑ jestliže je velikost vázané informace – záznamů příliš velká je výhodnější seřadit jen klíče s patřičnými odkazy na záznamy, které se v tomto případě nepřesouvají.
- ❑ Bez újmy na obecnosti budeme dále předpokládat, že řadíme pouze klíče.



# Vlastnosti metod řazení

---

## □ **sekvenční řazení**

- **sekvenčnost** je vlastnost, která vyjadřuje, že řadící algoritmus pracuje se vstupními údaji i s datovými meziprodukty v tom pořadí v jakém jsou lineárně uspořádány v datové struktuře
- Příklad: lineárně vázaný seznam

## □ **nesekvenční řazení**

- opakem sekvenčnosti je **přímý přístup** k jednotlivým položkám vstupní nebo pomocné struktury.
- Příklad: pole



# Vlastnosti metod řazení

---

## ☐ **Přirozenost**

- $T(\text{seřazená}) < T(\text{náhodně uspořádaná}) < T(\text{opačně uspořádaná})$
- $T$  – doba potřebná pro seřazení posloupnosti

## ☐ **Stabilita**

- algoritmus zachovává relativní uspořádání duplicitních klíčů souboru se shodnými klíči.
- nutná tam, kde řazení údajů podle klíče s vyšší prioritou nesmí porušit pořadí údajů se shodnými klíči vyšší priority, získané předcházejícím řazením množiny podle klíčů s nižší prioritou.

## ☐ **in situ** - metoda pracuje přímo v místě uložení řazených dat



# Vlastnosti metod řazení

*Příklad:* stabilita metod řazení, seřazeno podle příjmení

<b>Mocek</b>	<b>Tomasz</b>	<b>1000</b>	
<b>Musil</b>	<b>Martin</b>	<b>3000</b>	
<b>Navrkal</b>	<b>Pavel</b>	<b>5000</b>	
<b>Nečas</b>	<b>Ondřej</b>	<b>7000</b>	
<b>Nechvátal</b>	<b>Tomáš</b>	<b>4000</b>	
<b>Němeček</b>	<b>Marek</b>	<b>2000</b>	
<b>Neužil</b>	<b>Jiří</b>	<b>5000</b>	
<b>Novosad</b>	<b>Marek</b>	<b>2100</b>	
<b>Novotný</b>	<b>Jaroslav</b>	<b>2000</b>	
<b>Novotný</b>	<b>Tomáš</b>	<b>4500</b>	
<b>Obrdlík</b>	<b>Lukáš</b>	<b>5000</b>	
<b>Oczko</b>	<b>Jakub</b>	<b>2300</b>	



# Vlastnosti metod řazení

*Příklad:* stabilita metod řazení.

Osoby seřazené podle platu STABILNÍ METODOU				Osoby seřazené podle platu NESTABILNÍ METODOU		
Mocek	Tomasz	1000		Mocek	Tomasz	1000
Němeček	Marek	2000		Novotný	Jaroslav	2000
Novotný	Jaroslav	2000		Němeček	Marek	2000
Novosad	Marek	2100		Novosad	Marek	2100
Oczko	Jakub	2300		Oczko	Jakub	2300
Musil	Martin	3000		Musil	Martin	3000
Nechvátal	Tomáš	4000		Nechvátal	Tomáš	4000
Novotný	Tomáš	4500		Novotný	Tomáš	4500
Navrkal	Pavel	5000		Obrdlík	Lukáš	5000
Neužil	Jiří	5000		Neužil	Jiří	5000
Obrdlík	Lukáš	5000		Navrkal	Pavel	5000
Nečas	Ondřej	7000		Nečas	Ondřej	7000



# Vlastnosti metod řazení

---

## □ Složitost

(není obsahem tohoto předmětu, bude v IAL-Algoritmy)

- vyšetřování složitosti se *neodlišuje* od obvyklých postupů analýzy složitosti.
- prostorová a časová složitost označuje míru prostoru resp. času potřebnou k realizaci daného algoritmu,
- u řadicích algoritmů složitost označujeme jako funkci počtu  $n$  řazených prvků,
- zajímá nás limitní chování složitosti – tzv. **asymptotická složitost**,
- z hlediska prostorové složitosti jsou zajímavé zejména případy, kdy metoda pracuje přímo v místě uložení řazených dat (**in situ**) a nepotřebuje žádný (nebo jen minimální) přídavný prostor.



# Problematika porovnání českých řetězců

---

## □ Normy:

- ČSN ISO/IEC 14651- Informační technologie – Mezinárodní řazení a porovnání řetězců – Metoda pro porovnávání znakových řetězců a popis obecné šablony pro přizpůsobení řazení
- ČSN 97 6030 – Abecední řazení, 1994 - stanoví pravidla uspořádání abecedních sestav všeho druhu, obsahujících česká a cizojazyčná hesla. Dále stanoví zásady pro řazení číslic, různých znamének, značek a obrazců do abecedního řazení.



# Problematika porovnání českých řetězců

---

- ❑ Lexikografické porovnání řetězců
  - probíhá podle ordinálních čísel znaků
  - pouze pro angličtinu (bez diakritiky)

```
char str1[20], str2[20];  
...  
if (strcmp(str1, str2) > 0)  
{ ... }  
...
```

<0 – str1 je lexikograficky menší než str2  
=0 – shodné  
>0 – str1 je lexikograficky větší než str2





# Problematika porovnání českých řetězců

---

- ❑ Znaký s diakritikou mají v běžných variantách kódu ASCII [on line, cit. 2020-11-12] (ASCII - wikipedie) ordinální čísla větší než 127.

Příklad českého řazení:

Pozor! Písmena s diakritikou:

základní pořadí: č, ř, š, ž

řadí se jako bez diakritiky: á,ď,é,ě,í,ň,ó,ť,ú,ů,ý

- ❑ Základní způsoby řešení:
  - **konverze** řetězců do váhového kódu a porovnání výsledku normálním operátorem,
  - speciální **porovnávací funkce**.



# Problematika porovnání českých řetězců

## ❑ Triviální řešení:

- překódovat znaky podle nového uspořádání

Kód	Znak
1	A
2	B
3	C
4	Č
5	D
6	E
7	F
.	.

- ❑ V takovém uspořádání by bylo nutné sekvenčně vyhledat požadovaný znak a z tabulky zjistit jeho pořadí
- ❑ Jinými slovy: na místo znaku 'A' přijde znak s kódem 1 (není podstatné jaký je to znak) a provede se normální porovnání.



# Problematika porovnání českých řetězců

## □ Rychlá varianta:

```
const unsigned char XTABLE[256] =  
{ ['A']=1, ['B']=2, ... };  
char strCz[20], strComp[20];  
...  
int i;  
for(i = 0; i < strlen(strCz); i++)  
{  
    strComp[i] = XTABLE[(unsigned)strCz[i]];  
}  
strComp[i] = '\\0';  
...  
if(strcmp(strComp, str2) > 0)  
{...}
```



# Problematika porovnání českých řetězců

## ❑ Rychlá varianta:

```
const unsigned char XTABLE[256] =  
{ ['A']=1, ['B']=2, ... };
```

Znak	Kód
A	1
B	2
C	3
D	5
.	.
Č	4
.	.
Ř	21

Pozor! V případě indexování znakem s diakritikou je nutné použít přetypování,  
např.: `[(unsigned char) 'Č']`



# Problematika porovnání českých řetězců

---

- ❑ Tato varianta neřeší problém dvojznaku "CH".
- ❑ **Řešení**
  - zahrnout CH do tabulky jako jeden znak na správné místo (za 'H')
  - překódovat CH v řetězci na znak, který s ničím nekoliduje – například #, @, nebo některý z netisknutelných znaků – při výpisu nutno překódovat zpět



# Problematika porovnání českých řetězců

```
int i, j;
int length = strlen(strCz);
for(i = j = 0; i < length; i++, j++)
{
    if(strCz[i] == 'C' && i < (length-1)
        && strCz[i+1] == 'H')
    {
        strComp[j] = KodCh;
        i++;
    }
    else
        strComp[j] = XTABLE[(unsigned)strCz[i]];
}
strComp[j] = '\\0';
```



# Problematika porovnání českých řetězců

## ❑ Problém podpořadí:

- České uspořádání není dáno jen váhou znaku, ale i pořadím znaků ('Á' není vždy větší než 'A').

Správně:		Chybně:
Janoš		Janoš
Jánoš		Janoušek
Jánošík		Jánoš
Janoušek		Jánošík
Jánský		Jánský



# Problematika porovnání českých řetězců

---

- ❑ Uspořádání podle podpořadí
  - dáno nejprve pořadím znaků, podle první tabulky
  - až v případě shody se bere v úvahu druhá tabulka
- ❑ Možné řešení:
  - nezbytné víceprůchodové překódování
  - v prvním průchodu se provede konverze podle tabulky, která konvertuje znaky s podpořadím na stejný kód (např. 'A' i 'Á' stejně).
  - druhý průchod řetězcem provede kódování znaků podle tabulky s podpořadím a přidá pouze (překódované) znaky, kterých se to týká na konec překódovaného řetězce.





# Problematika porovnání českých řetězců

---

## □ Poznámky:

- existují obecné algoritmy vyhovující i složitějším jazykům než je čeština (čeština - pouze dvě úrovně).
- Podle charakteru řešeného problému, může být někdy účelné pracovat se všemi znaky jako s velkými nebo jako s malými písmeny. Pak lze do překódovací tabulky uložit stejné váhy pro malá i pro velká písmena!!
- Velká a malá písmena mají stejnou řadící platnost.



# Problematika porovnání českých řetězců

---

- ❑ Knihovny nástroje pro práci s češtinou
  - `int strcoll(const char *s1, const char *s2);`
    - `<string.h>`
    - Pro porovnávání lokalizovaných řetězců. Nejdříve je ale nutné nastavit lokalizované prostředí.
  - `char *setlocale(int category, const char *locale);`
    - `<locale.h>`
    - konstanta: `LC_COLLATE`
    - Můžete použít pro kontrolu zda jste seřadili vaše data správně česky.



# Problematika porovnání českých řetězců

---

## □ Poznámky k předchozím algoritmům

- Předchozí algoritmy používaly pomocné řetězce pro uložení výsledku překódování – nepracují **in situ**.
- Pro praktické použití **neefektivní**. (+ starosti s alokací paměti)
- Princip překódování pomocí tabulek lze využít při implementaci *porovnávací funkce*, která bude znaky z řetězců jeden po druhém **překódovávat a rovnou porovnávat**.



## Řazení podle více klíčů

---

*Příklad:* máme vytvořit dva seznamy osob z daného souboru s využitím jejich data narození.

V prvním seznamu budou osoby od nejstaršího k nejmladšímu, druhý seznam bude sloužit jako přehled o narozeninách.

Osoby se stejným datem narozenin (den, měsíc) budou seřazeny od nejstaršího k nejmladšímu.

Předpokládejme, že soubor neobsahuje dvě stejně staré osoby.



# Řazení podle více klíčů

---

Nechť je definován typ:

```
typedef struct datum
{
    unsigned short int rok,
    unsigned char mesic, den;
} TDatum;
```

Pro první seznam mají klíče sestupnou prioritu v pořadí ROK, MESIC, DEN, zatímco pro druhý seznam, mají sestupnou prioritu v pořadí MESIC, DEN, ROK.



# Řazení podle více klíčů

---

## □ Řešení č. 1

- Složená relace uspořádání.
- Funkce, která bude postupně srovnávat klíče se snižující se prioritou.
- Nerovnost klíčů vyšší priority → relace je rozhodnuta. Jinak se postupuje k porovnání klíčů s nižší prioritou.

## □ Výhody

- Jednoprůchodové řazení.
- Funguje i pro nestabilní řadící metody.



# Řazení podle více klíčů

---

## □ Řešení č. 2

- Postupné řazení podle jednotlivých klíčů.
- Nutno postupovat od nejmenší priority klíče k vyšší prioritě.
- Např. seznam podle stáří: nejprve podle DEN, pak podle MESIC, nakonec podle ROK

## □ Výhoda

- Jednodušší algoritmus

## □ Nevýhody

- Řadící metoda musí být stabilní!
- Víceprůchodové řešení → neefektivní



# Řazení podle více klíčů

*Příklad:* funkce porovnání dat pro řazení prvního seznamu

```
// Vrací záporné číslo, pokud je první osoba  
// (parametr) mladší než druhý, nulu, pokud  
// jsou shodné a kladné číslo, pokud je první  
// starší než druhý
```

```
int cmpDate(const TDatum *prvy,  
            const TDatum *druhy)  
{  
    if(prvy->rok < druhy->rok)  
        return 1;  
    else if(prvy->rok > druhy->rok)  
        return -1;
```





# Řazení podle více klíčů

pokračování

```
else { // roky jsou stejné
    if(prvy->mesic < druhy->mesic)
        return 1;
    else if (prvy->mesic > druhy->mesic)
        return -1;
    else { // roky i měsíce jsou stejné
        if(prvy->den < druhy->den)
            return 1;
        else if(prvy->den > druhy->den)
            return -1;
        else // obě data jsou shodná
            return 0;
    }
}
```



# Řazení bez přesunu položek

---

- ❑ Nejvýznamnější operace při řazení
  - porovnání dvou položek a přesun položky (např. výměna dvou položek).
- ❑ Přesun velkých položek → náročné, neefektivní.

*Příklad:* je dána struktura a pole nad ní:

```
typedef struct tpolozka
{
    TKlic klic;
    TData data;
} TPolozka;
TPolozka pole[N];
```



# Řazení bez přesunu položek

---

Pak lze vytvořit pomocné pole indexů (ukazatelů):

```
int ppole[N];
```

Na počátku se *ppole* inicializuje tak, aby každý prvek měl hodnotu svého indexu:

```
for(int i = 0; i < N; i++) ppole[i] = i;
```

Relace mezi prvky  $i_1$  a  $i_2$  bude mít v algoritmu řazení tvar např.:

```
pole[ppole[i1]].klic < pole[ppole[i2]].klic
```

a odpovídající přesun (výměna) bude mít tvar (změny pouze v *ppole*!!!):

```
int pom = ppole[i1];
```

```
ppole[i1] = ppole[i2];
```

```
ppole[i2] = pom;
```



# Řazení bez přesunu položek

			pole	před řazením
Index	data	klic	ppole	ppole
0		13	0	2
1		15	1	0
2		11	2	1
3		18	3	3

Po seřazení lze do výstupního pole v jednom průchodu zapsat seřazené pole cyklem:

```
for(int i = 0; i < N; i++)  
    vystPole[i] = pole[ppole[i]];
```



# Rozdělení algoritmů řazení

---

- ❑ Podle typu paměti v níž je řazená struktura uložena
  - **vnitřní** (interní) metody řazení (metody řazení polí) předpokládají uložení seřazované struktury v operační paměti a přímý (nesequenční) přístup k položkám struktury,
  - **vnější** (externí) řazení (metody řazení souborů) předpokládají sekvenční přístup k položkám seřazované struktury.
- ❑ Zvláštní skupina – indexsekvenční struktury (implementované na discích)
  - metody řazení často kombinují principy vnitřního i vnějšího řazení



# Rozdělení algoritmů řazení

---

[Sorting algorithm](#) [on line, cit. 2019-11-10]

## □ Podle způsobu využití klíčů

### ■ adresní řazení

- klíč je využit pro výpočet polohy ve výstupu
- stanovení absolutní polohy prvku  **$x$**  na základě hodnoty  **$Klíč(x)$**



# Rozdělení algoritmů řazení

---

## ■ asociativní řazení

- klíče jsou používány pro porovnání vzájemného pořadí prvků – využití hodnot klíčů ***x*** a ***y*** pro stanovení relativní polohy ve výstupu, tj. na základě ***Klíč(x) ≤ Klíč(y)***
- jednotlivé metody se liší ve způsobu dělení úseku a v místě využití porovnání.
  - Dělení úseku může být:
    - ✓ vyvážené (úsek rozdělíme na dva přibližně stejné díly),
    - ✓ nevyvážené (jeden z dílů je výrazně menší - obvykle jednoprvkový).



# Rozdělení algoritmů řazení

---

- ❑ podle řádu složitosti:
  - od  $O(n \log_2 n)$  do  $O(n^2)$
- ❑ podle základního principu řazení:
  - metody založené na principu **výběru** (selection) – postupný přesun největšího (nejmenšího) prvku do seřazené výstupní struktury,
  - metody založené na principu **vkládání** (insertion) – postupné zařazování prvku na správné místo ve výstupní struktuře,





# Rozdělení algoritmů řazení

---

- metody založené na principu **rozdělování** (partition) – rozdělování řazené množiny na dvě části tak, že prvky jedné jsou menší než prvky druhé,
- metody založené na principu **setřídění** (merging) – sdružování seřazených podmnožin do větších celků,
- metody **založené na jiných principech** – nesourodá skupina ostatních metod nebo kombinací metod.



# Rozdělení algoritmů řazení

---

Konvence po zápis řadících algoritmů:

1. Pro jednoduchost budeme ve všech algoritmech pracovat s rozměrem pole  $N$  (v praxi by bylo lepší rozměr předávat pomocí parametru funkce).

```
#define N 100  
int pole[N];
```

2. Pole je pouze polem klíčů.
3. Budeme řadit vzestupně.



# Metoda přímého výběru

---

([Straight selection-sort](#)) [on line, cit. 2018-11-12]

## **Algoritmus:**

1. Pole je rozděleno na seřazenou část a neseřazenou část,
2. V neseřazené části se nalezne minimum a vymění se s prvkem bezprostředně následujícím za seřazenou částí a tento prvek se do ní zahrne,
3. Na začátku má seřazená část délku nula, na konci má délku pole.



# Metoda přímého výběru

*Příklad:* je dána posloupnost čísel 11 3 27 8 50 22 12.  
Seřadte ji metodou přímého výběru.

11	3	27	8	50	22	12
3	11	27	8	50	22	12
3	8	27	11	50	22	12
3	8	11	27	50	22	12
3	8	11	12	50	22	27
3	8	11	12	22	50	27
3	8	11	12	22	27	50



# Metoda přímého výběru

---

```
void selectionSort(int pole[])
{
    for (int i = 0; i < N; i++) {
        int minI = i;          // index minima
        int min = pole[i];     // hodnota minima
        for (int j = i + 1; j < N; j++) {
            if (pole[j] < min) { // hledání minima
                minI = j;
                min = pole[j];
            }
        }
        pole[minI] = pole[i];
        pole[i] = min;
    }
}
```



# Metoda přímého výběru

## □ Vlastnosti:

■ Celkový počet porovnání:

$$C = \frac{N^2 - N}{2}$$

■ Počet přesunů:

$$M_{\max} \cong \frac{N^2}{4} + 3(N - 1)$$

■ Složitost metody přímým výběrem je tedy  $O(N^2)$

■ Metoda není stabilní.

■ Paměťová složitost:  $O(N)$

## □ Experimentální výsledky [ $10^{-2}$ s]

Opačně seřazená  
posloupnost

Náhodně uspořádaná  
posloupnost

n	128	256	512
OSP	64	254	968
NUP	50	212	744

Metoda je  
přirozená



# Algoritmy pro vyhledávání a řazení

---





# Kontrolní otázky

---

1. Uvedte klasifikaci vyhledávacích algoritmů.
2. Vysvětlete princip sekvenčního vyhledávání v poli (v poli se zarážkou, v seřazeném poli, v seřazeném poli se zarážkou).
3. Vysvětlete princip binárního vyhledávání
4. Vysvětlete, co znamená když metoda řazení je: sekvenční, přirozená, stabilní, pracuje in situ.
5. Vysvětlete princip řazení podle více klíčů
6. Vysvětlete princip řazení bez přesunu položek.





# Úkoly k procvičení

---

1. Realizujte algoritmy sekvenčního vyhledávání v neseřazeném poli s údaji o osobách, které budou obsahovat tyto údaje (JMÉNO, PŘÍJMENÍ, ULICE, MĚSTO). Uživatele nechte zvolit, podle jaké položky chce vyhledávat.
2. Realizujte vyhledávání záznamu ve stejném formátu, jako v předešlém případě, ale předpokládejte, že jsou seřazeny podle příjmení. V tomto případě nevyhledávejte podle jiných položek.
3. Pomocí stejné množiny dat srovnejte výsledky předchozích dvou programů.