

# 11 Algoritmy pro vyhledávání a řazení

Kreslíková, 2. 11. 2023

11 Algoritmy pro vyhledávání a řazení .....	1
11.1 Úvod .....	1
11.2 Algoritmy pro vyhledávání.....	2
11.2.1 Klasifikace vyhledávacích algoritmů .....	2
11.2.2 Sekvenční vyhledávání .....	3
11.2.2.1 Sekvenční vyhledávání v poli se zarážkou.....	4
11.2.2.2 Sekvenční vyhledávání v seřazeném poli .....	4
11.2.2.3 Sekvenční vyhledávání v seřazeném poli se zarážkou.....	4
11.2.3 Nesequenční vyhledávání .....	5
11.2.3.1 Binární vyhledávání .....	5
11.3 Algoritmy pro řazení .....	6
11.3.1 Vlastnosti metod řazení .....	6
11.3.2 Problematika porovnání českých řetězců .....	7
11.3.3 Řazení podle více klíčů .....	10
11.3.4 Řazení bez přesunu položek .....	11
11.3.5 Rozdělení algoritmů řazení.....	12
11.3.6 Metoda přímého výběru (Selection-sort).....	12
11.3.7 Metoda bublinového řazení (Bubble-Sort) .....	14
11.3.8 Řazení na principu vkládání (Insert-Sort) .....	16
11.4 Shrnutí .....	17
11.5 Úlohy k procvičení .....	18
11.6 Kontrolní otázky .....	18

Tato publikace je určena výhradně jako podpůrný text pro potřeby výuky. Bude užita výhradně v přednáškách výlučně k účelům vyučovacím či jiným vzdělávacím účelům. Nesmí být používána komerčně. Bez předchozího písemného svolení autora nesmí být kterákoliv část této publikace kopírována nebo rozmnožována jakoukoliv formou (tisk, fotokopie, mikrofilm, snímání skenerem či jiný postup), vložena do informačního nebo jiného počítačového systému nebo přenášena v jiné formě nebo jinými prostředky. Veškerá práva vyhrazena © Jitka Kreslíková a kol., Brno 2023

## 11.1 Úvod

Vyhledávání a řazení dat jsou dvě nejtypičtější algoritmické úlohy. V dnešní době jsou totiž počítače používány mnohem častěji pro ukládání velkých objemů dat a vyhledávání mezi nimi než pro vědeckotechnické výpočty. Tyto úlohy jsou zároveň pěknými algoritmickými cvičeními, které nám pomohou upevnit znalosti získané v předchozích kapitolách.

V této kapitole se podíváme nejprve na vyhledávací algoritmy. Rozdělíme si je podle algoritmického přístupu k vyhledávání i podle způsobu uložení a zacházení s daty. Probereme různé varianty sekvenčního vyhledávání, dále se podíváme na algoritmus binárního vyhledávání.

Ve druhé části této kapitoly se podíváme na řadící algoritmy. Kromě jednoho základního algoritmu řazení se podíváme blíže také na vlastnosti řadících algoritmů, abychom je mohli vzájemně srovnávat. Abychom byli schopni řešit i reálné úlohy, zaměříme se také na praktické problémy spojené s řazením a vyhledáváním dat. Konkrétně jde o problematiku porovnávání českých textů a řazení podle více klíčů.

## 11.2 Algoritmy pro vyhledávání

Vyhledávání a řazení má stejný cíl – urychlit vyhledávání údajů ve velkých objemech dat. Vyhledávání (searching) se zabývá algoritmy umožňujícími co nejefektivnější nalezení hledaného údaje. Pro rychlé vyhledávání je nutné činit jisté kroky již při ukládání dat. Problematika vyhledávání se proto někdy nazývá ukládání a vyhledávání dat (Data Storage and Retrieval). Pokročilé algoritmy pro rychlé vyhledávání lze dnes nalézt zejména v oblasti databázových systémů.

### 11.2.1 Klasifikace vyhledávacích algoritmů

Ne každý vyhledávací algoritmus se hodí pro všechny typy prohledávaných dat. Klasifikace těchto algoritmů podle různých kritérií zjednoduší volbu vhodného algoritmu pro konkrétní data.

Podle místa uložení dat

- interní – data v operační paměti
- externí – data na disku
- kombinované – nalezení bloku na disku a dohledání v paměti

Podle principu

- sekvenční vyhledávání – sekvenčně se prochází prohledávaná struktura.
- indexsekvenční vyhledávání – přímým přístupem se nalezne blok a v něm se sekvenčně dohledá ([ISAM](#) – *Indexed Sequential Access Method*) [on line, cit. 2022-12-05].
- nesequenční vyhledávání v seřazeném poli – index prvku se nalezne rychleji než sekvenčním průchodem.
- vyhledávání v seřazených stromech – použití speciálně organizovaných struktur pro uložení prohledávaných dat.
- vyhledávání v tabulkách s rozptýlenými položkami (hash table) – index hledaného prvku se „vypočte“ speciální funkcí (tento princip se často kombinuje s indexsekvenčním přístupem).

Podle práce s klíči

- původní klíče
- transformované klíče – vedou k tabulkám s rozptýlenými položkami.

#### Jednorozměrné vyhledávání

- **adresní** vyhledávací algoritmy využívají jednoznačného vztahu mezi hodnotou klíče prvku a umístěním prvku ve struktuře reprezentující vyhledávací prostor  $S$  (přímý přístup k prvkům pole pomocí indexů)
- **asociativní** vyhledávací algoritmy porovnávají prvky (relativní hodnota vzhledem k ostatním prvkům) při hledání prvku  $x \in S$  se využívají relace (porovnávání) mezi prvky struktury reprezentující  $S$ . Mezi prvky musí existovat relace uspořádání. Většinou požadujeme, aby byl prostor prvků uspořádán podle této relace.

#### Vícerozměrné vyhledávání

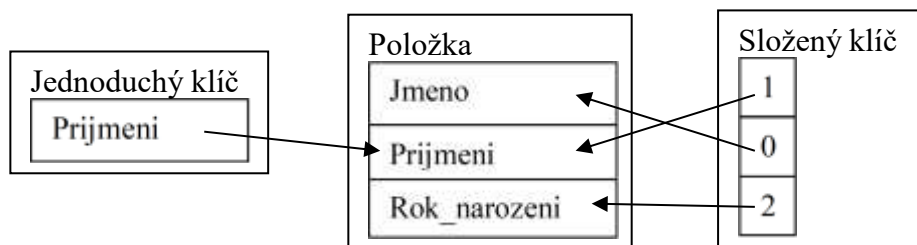
Vícerozměrné vyhledávání se od jednorozměrného vyhledávání liší se v tom, že nevyhledáváme jen podle jednoho klíče, ale podle více klíčů. Tomuto požadavku se musí přizpůsobit datové struktury a algoritmy.

Samotné dotazy se dají rozdělit do tří kategorií:

- dotazy na úplnou shodu – dotazy požadující shodu na všech klíčích.
- dotazy na částečnou shodu – dotazy požadující shodu jen na některých složkách klíče.
- dotazy na intervalovou shodu – dotazy požadující, aby klíč záznamu byl v určeném intervalu.

## Složený klíč, řazení podle složeného klíče

Pojem složený klíč, respektive řazení podle složeného klíče, používáme tehdy, kdy pro porovnávání položek slouží více klíčů. Každý takový klíč (respektive složka složeného klíče) může obsahovat data různých datových typů. Operaci porovnání dat podle složeného klíče potom musíme vytvořit zřetěžením porovnávacích operací nad jednotlivými složkami klíče ve správném pořadí.



Obrázek 1.: Ilustrace rozdílu mezi přístupem s jednoduchým a složeným klíčem.

### 11.2.2 Sekvenční vyhledávání

Základní princip sekvenčního vyhledávání je, že se sekvenčně prochází všechny hodnoty až do nalezení hledané hodnoty nebo dosažení konce dat. Vhodnou datovou strukturou, která obecně umožňuje efektivní ukládání a vyhledávání informací je tabulka. Jednoduchou implementací tabulky je pole, jehož prvky jsou záznamy obsahující klíč a hodnotu. Pole může být uspořádané nebo neuspořádané. Neuspořádané pole znamená, že prvky jsou ukládány bez ohledu na hodnotu klíče.

Vyhledávání prvku se zadaným klíčem znamená postupné prohledání pole (sekvenční zpracování).

Nechť jsou dány datové typy :

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

Nad typem **typKlice** je definována relace ekvivalence. V dalších algoritmech budeme používat funkce **isEqual** a **isLess**, které podle hodnoty klíče a dat budou vracet hodnotu typu **bool**.

*Poznámka1:* je nutné si uvědomit, že porovnávání určitých typů dat není triviální operací, zvláště pokud používáme složené klíče.

*Poznámka2:* aby mohly být následující ukázky dostatečně názorné, obsahuje struktura **TPoložka** samostatnou složku **klic**. V praktických aplikacích ovšem většinou tvoří klíč samotné datové složky struktury. U složených klíčů je klíč tvořen několika (nebo všemi) složkami struktury současně a nějakou vhodnou datovou strukturou, která udává pořadí těchto složek v klíči.

Další deklarace

```
#define N 100 // delka pole
TPoložka pole[N];
```

Pole je použito pro implementaci abstraktního datového typu (ADT) tabulka a

```
typKlice k;
```

má hodnotu vyhledávaného klíče; pak sekvenční vyhledávání v poli realizuje následující algoritmus:

*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?

### 11.2.2.1 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).

*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 na místo rušeného a snížit velikost pole.

### 11.2.2.2 Sekvenční vyhledávání v seřazeném poli

Obecně je vyhledávání v seřazeném poli mnohem jednodušší a efektivnější operace, než vyhledávání v neseřazeném poli. Zvýšení efektivity se projeví i při jednoduchém sekvenčním vyhledávání.

*Příklad:* Sekvenční vyhledávání v seřazeném poli.

```

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

```

### Dynamické vlastnosti vyhledávání v seřazeném poli

Při vkládání a rušení prvku je nutné zachovat uspořádanost – může jít o „drahou“ operaci.

Do určitého počtu nových prvků je lepší přidávat na konec a neuspořádanou část projít sekvenčně.

### 11.2.2.3 Sekvenční vyhledávání v seřazeném poli se zarážkou

*Příklad:* Sekvenční 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

```

### 11.2.3 Nesequenční vyhledávání

Nesequenční vyhledávání se od sekvenčního liší tím, že nedochází k průchodu všemi prvky v pořadí, v jakém jsou uloženy v tabulce. S výjimkou v podobě hašovací tabulky je nutné, aby tabulka, ve které chceme takto vyhledávat byla seřazena podle údajů, které používáme jako vyhledávací klíč.

#### 11.2.3.1 Binární vyhledávání

Nechť 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 :

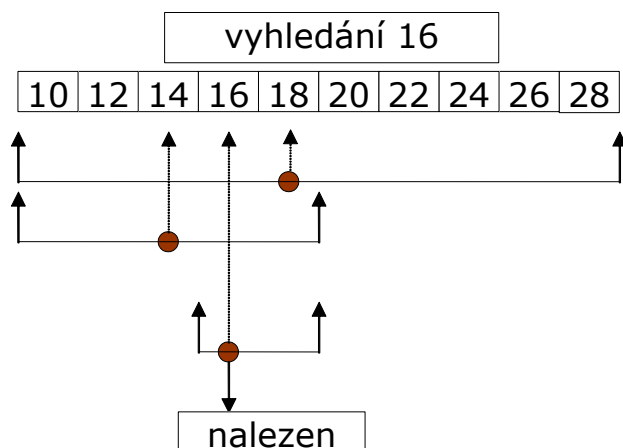
$$k \geq \text{pole}[0].\text{klic} \text{ a } k \leq \text{pole}[N-1].\text{klic}$$

*Poznámka:* zde uváděné operace porovnávání jsou jenom zjednodušující abstrakcí (viz poznámky o složených klíčích výše).

#### 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 vyhledávaný klíč větší postupuje se porovnáváním prostředního prvku v pravé polovině původního pole. Vyhledávání končí neúspěchem v případě, že prohledávaná část pole je prázdná (její levý index je větší než pravý).

*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
}
```

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

## 11.3 Algoritmy pro řazení

### Obecná formulace problému řazení

Je dána neprázdná množina  $A = \{a_1, a_2, \dots, a_n\}$ . Je potřebné najít permutaci  $\pi$  těchto  $n$  prvků, která zobrazuje danou posloupnost do neklesající 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**.

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

### 11.3.1 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.

#### Nesekvenční řazení

Opakem sekvenčnosti je přímý přístup k jednotlivým položkám vstupní nebo pomocné struktury. O algoritmu, který nepracuje sekvenčně říkáme, že je nesekvenční.

#### Přirozenost

Přirozenost řazení je vlastnost algoritmu, která vyjadřuje, že doba potřebná k řazení již seřazené množiny údajů je menší než doba pro seřazení náhodně uspořádané množiny a ta je menší než doba pro seřazení opačně seřazené množiny údajů.

#### Stabilita

Stabilita řazení je vlastnost řazení, která vyjadřuje, že algoritmus zachovává relativní uspořádání duplicitních klíčů souboru se shodnými klíči.


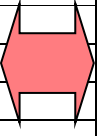
Stabilita je nutná tam, kde se vyžaduje, aby se při řazení údajů podle klíče s vyšší prioritou neporušilo pořadí údajů se shodnými klíči vyšší priority, získané předcházejícím řazením množiny podle klíčů s nižší prioritou.

*Příklad:* Stabilita metod řazení, osoby seřazené podle příjmení.

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

Tabulka 1.: Osoby seřazené podle příjmení

*Příklad:* Stabilita metod řazení, osoby seřazené podle platu.

Osoby seřazené podle platu				Osoby seřazené podle platu		
STABILNÍ METODOU				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

Tabulka 2.: Osoby seřazené podle platu

Složitost řadicích algoritmů (není obsahem tohoto předmětu)

Vyšetřování složitosti řadicích algoritmů se *neodlišuje* od obvyklých postupů. 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.

### 11.3.2 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.

```
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
```

Znaky s diakritikou mají v běžných variantách [kódu ASCII](#) [on line, cit. 2018-11-10] ordinální čísla větší než 127.

Základní způsoby řešení:

- konverze řetězců do váhového kódu a porovnání výsledku normálním operátorem
- speciální porovnávací funkce

Triviální řešení:

Překódovat znaky podle nového uspořádání tak, aby platilo např.:

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.

### Rychlá varianta

*Příklad: Překódování řetězce.*

```
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)
{...}
```

Kód	Znak
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']`

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.



*Příklad: Přiřazení odpovídajícího kódu písmenu CH.*

```
.
.
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';
.
.
```

### 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

Uspořádání podle podpořadí je dáno nejprve pořadím znaků, kterých se to týká, bez diakritiky a až v případě shody se bere v úvahu diakritika. České uspořádání textových řetězců je definováno jako dvouprůchodové, kdy se nejprve porovnává pomocí první tabulky (některé znaky, např. A, Á zde mají stejnou váhu), pokud jsou slova po tomto porovnání shodná, musí se udělat ještě jeden průchod, kdy se bere v úvahu druhá tabulka (zde už je definováno pořadí všech písmen s diakritikou).

### Možné řešení

Je nezbytné víceprůchodové překódování podle několika tabulek. 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.

*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.

### Knihovní nástroje pro práci s češtinou

```
<string.h>
int strcoll(const char *s1, const char *s2);
```

Pro porovnávání lokalizovaných řetězců. Nejdříve je ale nutné nastavit lokalizované prostředí.

```
<locale.h>
char *setlocale(int category, const char *locale);
konstanta: LC_COLLATE
```

*Poznámka k efektivitě:* předchozí algoritmy používaly pomocné řetězce pro uložení výsledku překódování – nepracují in situ. To si ovšem můžeme dovolit pouze u textových řetězců omezené délky. Pro praktické použití s řetězci obecné délky je to neefektivní, protože nám přibudou starosti se správným alokováním a uvolňováním paměti. Princip překódování pomocí tabulky je možné využít při implementaci porovnávací funkce, která bude znaky z řetězců jeden po druhém překódovávat a rovnou porovnávat.

### 11.3.3 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 narození budou seřazeny od nejstaršího k nejmladšímu. Předpokládejme, že soubor neobsahuje dvě stejně staré osoby.

Necht' 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.

K řešení vedou dva přístupy:

1. Vytvoření složené relace uspořádání, která může mít tvar funkce, ve které se budou postupně srovnávat klíče se snižující se prioritou. Dojde-li k nerovnosti u klíčů vyšší priority, je relace rozhodnuta. Jinak se postupuje k porovnání klíčů s nižší prioritou. Pomocí takové relace lze seřadit osoby podle stáří v jednom řazení. Funguje i pro nestabilní řadící metody.
2. Postupné řazení podle jednotlivých klíčů se dosáhne téhož seřazení, bude-li se postupovat v řazení od nižší priority klíče k vyšší prioritě. To znamená, že seznam osob podle stáří získáme, budeme-li daný soubor postupně řadit podle klíče DEN, pak podle klíče MESIC, a nakonec podle klíče ROK. Řadící metoda musí být stabilní! Nevýhodou je, že musí být použito víceprůchodové řešení, které je neefektivní.

Jiná možnost je nalézt takové kódování, aby bylo možné použít přímé porovnání.

*Příklad:* Funkce pro 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;
    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 shodné
        return 0;
}
}
}

```

### 11.3.4 Řazení bez přesunu položek

Nejvýznamnějšími operacemi každého algoritmu řazení jsou porovnání dvou položek a přesun položky (např. výměna dvou položek). Zabírá-li položka, která se má přesunout, větší paměťový prostor, pak její přesun je časově náročný.

*Příklad:* Je dána struktura a pole nad ní.

```

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

```

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;

```

Index	Data	Klic	ppole	ppole
1		13	1	3
2		15	2	1
3		11	3	2
4		18	4	4

**Obrázek 2.:** Ilustrace obsazení polí při řazení bez přesunu položek.

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]];

```

### 11.3.5 Rozdělení algoritmů řazení

Existuje více různých kritérií.

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 (nesekvenční) přístup k položkám struktury.
- vnější (externí) řazení (metody řazení souborů) předpokládají sekvenční přístup k položkám seřazované struktury.

Zvláštní skupinu tvoří indexsekvenční struktury (implementované na discích). Metody řazení často kombinují principy vnitřního i vnějšího řazení.

Podle způsobu využití klíčů

- adresní řazení (klíč je využit pro výpočet polohy ve výstupu - stanovení absolutní polohy prvku  $x$  na základě hodnoty  $Klíč(x)$  )
- 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  $Klíč(x) \leq Klíč(y)$  )

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).

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.
- 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.

### Konvence po zápis řadících algoritmů

Pro jednoduchost budeme ve všech algoritmech pracovat s rozměrem pole  $N$  (v praxi by bylo lepší rozměr předávat pomocí parametru funkce). Pole je pouze polem klíčů. Budeme řadit vzestupně.

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

### 11.3.6 Metoda přímého výběru ([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. Krok 2 opakujeme tak dlouho, dokud neseřazená část obsahuje více než jeden prvek. Jinak algoritmus končí.

Na začátku má seřazená část délku nula, na konci má délku pole.

*Příklad:* Je dána posloupnost čísel 11 3 27 8 50 22 12. Seřadíte 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

```
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;
    }
    return;
}
```

### Vlastnosti metody přímého výběru

- Celkový počet porovnání:  $C = \frac{N^2 - N}{2}$
- Počet přesunů:  $M_{\max} \cong \frac{N^2}{4} + 3(N - 1)$
- Časová složitost metody přímým výběrem je tedy  $O(N^2)$ .
- Paměťová složitost je  $O(N)$
- Metoda není stabilní.

Experimentální výsledky [10<sup>-2</sup>s], OSP - Opačně Seřazená Posloupnost. NUP - Náhodně Uspořádaná Posloupnost.

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

- Metoda je přirozená.

### 11.3.7 Metoda bublinového řazení ([Bubble-Sort](#))

[on line, cit. 2022-12-05]

Algoritmus:

1. Posloupnost rozdělíme na dvě části, seřazenou a neseřazenou. Seřazená část je prázdná,
2. Postupně porovnáme všechny sousední prvky v neseřazené části a pokud nejsou v požadovaném pořadí, prohodíme je,
3. Krok 2 opakujeme tak dlouho, dokud neseřazená část obsahuje více než jeden prvek. Jinak algoritmus končí.

*Příklad:* Je dána posloupnost čísel 11 27 8 50 22 3 12. Setřídíte ji metodou bublinového řazení.

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

*Příklad:* Modifikace kroku 3 (bublinové řazení): Pokud nastala v kroku 2 alespoň jedna výměna, došlo ke zmenšení neseřazené části o jeden prvek, pokračujeme bodem 2. Jinak je řazení ukončeno.

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

```
void bubbleSort(int pole[])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = N - 1; j > i; j--)
        {
            if (pole[j - 1] > pole[j])
            { // výměna prvků
                int x = pole[j - 1];
                pole[j - 1] = pole[j];
                pole[j] = x;
            }
        }
    }
    return;
}
```

## Vlastnosti metody bublinového řazení

- Počet porovnání v jednotlivých krocích bude  $C_i = N-i$ .

$$M = \sum_{i=2}^N M_i = \frac{3(N^2 - N)}{2}$$

- Počet přesunů v i-tém kroku je  $M_i = 3(N-i)$ , celkem:
- Časová složitost metody bublinového řazení je tedy  $O(N^2)$ .
- Paměťová složitost je  $O(N)$ .
- Metoda je stabilní.

Experimentální výsledky [10<sup>-2</sup>s], OSP - Opačně Seřazená Posloupnost. NUP - Náhodně Uspořádaná Posloupnost.

n	256	512
NUP	338	1562
OSP	558	2224

- Metoda je přirozená.

Metoda bublinového řazení je ze všech metod nejrychlejší pro uspořádané pole, v ostatních případech je nejhorší!!! Doporučuje se ji používat pouze v odůvodněných případech. Je vhodná pro řazení seznamů.

*Poznámka:* používá se v knihovnách pro řazení krátkých polí – méně než 10 prvků.

## Varianty Bubble-Sort

Ripple-Sort - pamatuje si, kde došlo v minulém průchodu k první výměně a začíná až z tohoto místa.

Shaker-Sort - prochází oběma směry a "vysouvá" nejmenší na začátek a největší na konec.

Shuttle-Sort - dojde-li k výměně, algoritmus se vrací s prvkem zpět, pokud dochází k výměnám. Pak se vrátí do místa, odkud se vracel a pokračuje dál.

Varianty lze kombinovat. Varianty jsou pěkná algoritmická cvičení, zlepšení složitosti jsou však pouze kosmetická.

Srovnání experimentálních výsledků, SP - Seřazená Posloupnost. NUP - Náhodně Uspořádaná Posloupnost. OSP - Opačně Seřazená Posloupnost.

n = 256	Bubble-Sort	Shaker-Sort
SP	2	2
NUP	388	340
OSP	588	588

### 11.3.8 Řazení na principu vkládání ([Insert-Sort](#))

[on line, cit. 2022-12-05]

Algoritmus:

1. Pole je rozděleno na seřazenou a neseřazenou část.
2. V seřazené části se nalezne pozice, na kterou přijde vložit první prvek z neseřazené části a od této pozice až do konce seřazené části se prvky odsunou.
3. Krok 2 opakujeme tak dlouho, dokud neseřazená část obsahuje nějaký prvek. Jinak algoritmus končí.

Na začátku má seřazená část délku jedna.

*Příklad:* Je dána posloupnost čísel 11 3 27 8 50 22 12. Seřadíte ji metodou vkládání.

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

```
void insertSort(int pole[])
{
    for (int i = 1; i < N; i++)
    {
        int x = pole[i];
        int j = i;
        while ((j > 0) && (x < pole[j - 1]))
        { // posuv prvků o jedničku doprava
            pole[j] = pole[j - 1];
            j--;
        }
        pole[j] = x;
    }
    return;
}
```

#### Vlastnosti metody řazení na principu vkládání

- Nejhorší případ nastává, jestliže zdrojová posloupnost je uspořádána v opačném pořadí.
- Počet porovnání pro  $i$ -tý prvek bude  $C_i = i - 1$ . Počet přesunů  $M_i = C_i + 2$ .
- Celkový počet porovnání  $C_{\max}$  a přesunů  $M_{\max}$  pro nejhorší případ bude:

$$C_{\max} = \sum_{i=2}^N C_i = \frac{N^2 - N}{2} \qquad M_{\max} = \sum_{i=2}^N M_i = \frac{N^2 + 3N - 4}{2}$$

- Časová složitost metody řazení na principu vkládání je tedy  $O(N^2)$ .
- Paměťová složitost je  $O(N)$ .
- Metoda je stabilní.
- Metoda je přirozená.



## 11.4 Shrnutí

Po absolvování a praktickém procvičení této kapitoly je čtenář schopen definovat úlohy vyhledávání a řazení dat. Umí klasifikovat vyhledávací algoritmy podle ukládání dat, podle principu a způsobu práce s klíči. Dále dokáže popsat principy jednorozměrného vyhledávání, ale i vícerozměrného vyhledávání podle složeného klíče. Čtenář nyní dokáže formulovat tři varianty algoritmu sekvenčního vyhledávání. Z algoritmů nesequenčního vyhledávání je schopen používat algoritmus binárního vyhledávání.

Čtenář nyní také dokáže popsat princip a v jazyce C naprogramovat řadící algoritmus přímého výběru. Čtenář je nyní schopen tento algoritmus realizovat i pro pole struktur, která je schopen řadit podle více klíčů. Čtenář také dokáže popsat algoritmus porovnávání českých řetězců podle české normy ČSN 97 6030 a použít jej jak při řazení, tak při vyhledávání.

## 11.5 Úlohy 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 záznamy ve stejném formátu, jako v předešlém případě, ale předpokládejte, že jsou seřazeny podle příjmení. V tomto případě nevyhledávejte podle jiných položek.
3. Vytvořte program pro řazení číselné posloupnosti výše probraným algoritmem pro řazení. Výběr algoritmu řazení provádějte pomocí parametru z příkazové řádky.
4. Modifikujte předchozí program tak, aby byl schopen řadit místo číselné posloupnosti posloupnost slov, tedy textových řetězců podle anglické abecedy.
5. Předchozí programy aplikujte na posloupnost seřazenou, opačně seřazenou a náhodně uspořádanou. Zaznamenávejte získané výsledky a diskutujte je ve vztahu k dalším metodám řazení. Jsou nějaké rozdíly mezi výsledky řazení číselných a textových posloupností?

## 11.6 Kontrolní otázky

1. Uveďte 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.
7. Proč je řadící metoda selection-sort nestabilní?
8. Vysvětlete metodu řazení založenou na principu přímého výběru (Straight selection-sort). Uveďte její vlastnosti.
9. Vysvětlete metodu bublinového řazení (Bubble-sort). Uveďte její vlastnosti.
10. Vysvětlete metodu řazení založenou na principu vkládání (Insert-sort). Uveďte její vlastnosti.