Vyhledávání v tabulkách

- **Tabulkou** nazveme množinu položek identifikovatelných hodnotou přístupového (identifikačního) **klíče** (*key*, **ID** – *identificator*).
  - Ve vodorovném směru se jedná o **heterogenní pole**, tzn. že každá položka je složena z několika údajů, které mohou být různých (datových) typů.
  - Ve svislém směru se jedná o **homogenní pole** – pole položek identifikovatelných pomocí klíče.

	klíč (ID)	údaj 1	údaj 2	…	údaj *m*
položka 1					
položka 2					
…					
položka *n*					

- Přístup do tabulky provádíme pomocí klíče a nadále budeme dodržovat následující konvence ve značení:
  - ***k*** – **identifikační klíč**, tj. klíč, kterým položku identifikujeme
  - A_k – **adresní klíč**, tj. adresa položky, kterou je ve většině případů index (pořadí) položky.

- **Implementace tabulky**

```
type    T_klic = integer;   {například}

        Polozka = record
             Klic : T_klic;
             Udaj1 : string[délka];
             Udaj2 : real ;
             …
        end;

        Tabulka = array [1..n] of Polozka
```

  - Klíč může být jakéhokoliv datového typu (`T_klic`), obvykle se však používá celé číslo (*integer*) nebo řetězec znaků (*string*).

Hledání v tabulce

- **Parametry hledání** v tabulce:
  - ***S*** – **délka hledání položky**, tj. počet položek, které je nutno zkoumat
  - ***T*** – **průměrná délka hledání** definovaná vztahem

$$T = \frac{1}{p}\sum_{i=1}^{p} S_i$$

kde p je počet přístupů do tabulky

- ***A*** – **průměrná délka prohledávání** za předpokladu rovnoměrného přístupu

$$A = \frac{1}{n}\sum_{i=1}^{n} S_i$$

kde n je počet obsazených položek tabulky

- ***P*** – tzv. **plnění tabulky** (podíl obsazených položek) definované vtahem:

$$P = \frac{n}{a}$$

kde n je počet obsazených míst a a počet všech alokovaných míst tabulky

Rozdělení tabulek

- Podle **počtu položek** můžeme tabulky rozdělit na:
  - **Pevně definované tabulky** (**LUT** – *Look Up Table*)
    - počet položek je konstantní
  - **Tabulky s proměnným počtem položek**
    - počet položek je proměnný

- Podle **způsobu hledání** rozdělujeme tabulky na:
  - **Tabulky s uspořádanými položkami**
    - **primární klíč** - klíč, podle kterého je tabulka uspořádána.
    - **sekundární klíč** - všechny ostatní klíče
  - **Tabulky s neuspořádanými položkami** (i tabulky s rozptýlenými položkami)

- Tabulky lze podle **způsobu organizace** rozdělit na:
  - **Tabulky s přímým přístupem**
    - $k \rightarrow A_k$ je prosté zobrazení, každá položka v tabulce má své místo jednoznačně určené hodnotou A_k přímo odvozenou z k
    - *Příklad*: Tabulka odkazů na oddíly slovníku (A, B, C, …)
  - **Obyčejné vyhledávací tabulky**
    - hledáme prostřednictvím klíče, pokud najdeme shodu, našli jsme danou položku
  - **Tabulky s rozptýlenými položkami** (*hash tables*)
    - $k \rightarrow A_k$ se nazývá rozptylovací funkce, na kterou jsou kladeny speciální požadavky

Tabulky s přímým přístupem

- Protože je $k \rightarrow A_k$ prosté zobrazení, je $S = T = A = 1$.

- Většinou $k = A_k$, proto není třeba klíč uchovávat a uchováváme jen data (údaje) – tomu se říká **degenerovaná tabulka**

- **Nevýhoda**: velikost je dána rozsahem klíče, v praxi je tedy často příliš velká. Navíc, pokud nevyužijeme všechny klíče, jsou v tabulce nevyužité díry, což je plýtvání pamětí

Vyhledávací tabulky

- Vyhledávání položky se provádí porovnáváním podle klíče.

- **Pořadí položek** může být
  - **definované**
  - **náhodné**

- **Strategie vyhledávání**
  - **Sekvenční**
    - Bereme postupně prvky v takovém pořadí, v jakém jsou v tabulce uloženy, porovnáváme klíče s hledanou hodnotou, a to tak dlouho, dokud nenajdeme shodu. Platí:

$$A = \frac{1}{2}(n+1)$$

      položky mohou být v tabulce uloženy v libovolném pořadí

    - **Implementace sekvenčního vyhledávání**

```
function Sekvencni(T:Tabulka;Hled_klic:T_klic):integer;
var i:integer;
begin
   i:=1;
   while (i<=n) and (T[i].Klic<>Hled_klic) do i:=i+1;
   if i<=n then Sekvencni:=i
   else Sekvencni:=0   {Klíč nenalezen}
end;
```

  - **Půlením (binární, logaritmické)**
    - Lze použít, pokud je tabulka seřazená podle klíče. Porovnáním klíče prvku v polovině tabulky s hledaným rozhodneme, v které polovině tabulky se hledaný prvek nachází. V této polovině pak opět zkusíme prvek uprostřed, atd. …

$$A \approx \log(n)$$

    - **Implementace binárního vyhledávání**

```
function Binarni(T:Tabulka;Hled_klic:T_klic):integer;
var i,k:integer;
begin
   k:=n;I:=n;
   while (K>0) and (i<=n) and (T[i].klic<>Hled_klic) do
   begin
        k:=k div 2;
        if Hled_klic<T[i].klic then i:=i-k
        else i:=i+i
   end;
   if (K>0)and(I<=K) then Binarni:=i  else Binarni:=0
end;
```

Pomocí Fibonacciho posloupnosti

- Myšlenka je stejná jako u binárního vyhledávání, jen zkoumané prvky nevolíme uprostřed, ale interval dělíme v poměru Fibonacciho čísel
- Operační složitost je také řádu log(n), ale pro prvky na začátku tabulky je vyhledávání rychlejší
- Pro efektivní hledání se snažíme, aby měla tabulka F_n-1 položek, kde F_n je určité Fibonacciho číslo
- Fibonacciho čísla tvoří posloupnost definovanou následujícím rekurentním vztahem:

$$F_0 = 0 \qquad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \qquad \forall n \in N, n \geq 2$$

tedy posloupnost tvoří čísla 0,1,1, 2, 3, 5, 8, 13, 21, 34, 55, …

Hledání podle sekundárního klíče

- převádíme na hledání podle primárního klíče
- implementace **invertovaným souborem** (indexový soubor) - jedná se ve skutečnosti o tabulku seřazenou podle hledacího klíče, jejíž data jsou primární klíč (nebo přímo index v původní tabulce). V praxi je tato tabulka často degenerovaná a vyhledávací klíč se neudává, stačí, že je podle něj invertovaný soubor seřazen.

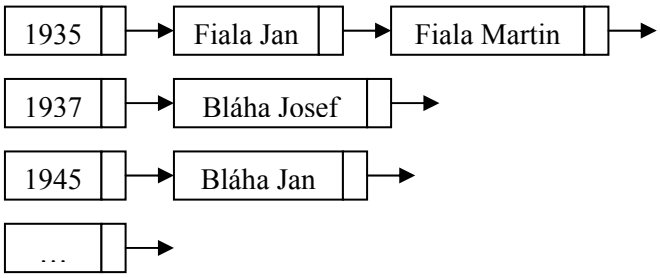
primární klíč	data (sekundární klíč)
Bláha Jan	1945
Bláha Josef	1937
Fiala Jan	1935
Fiala Martin	1935
Fiala Petr	1956
Zubák Jiří	1948

→

invertovaný soubor	
1935	Fiala Jan
1935	Fiala Martin
1937	Bláha Josef
1945	Bláha Jan
1948	Zubák Jiří
1956	Fiala Petr

- implementace **invertovaným seznamem** - invertovaný soubor realizovaný jako spojový seznam

nebo



- **Vícenásobný přístup**
  - Vyhledávání v tabulce podle více klíčů (vícerozměrné vyhledávání) realizujeme vícenásobným přístupem, například hledáme postupně nejdříve podle příjmení, pak podle jména, atd.

- **Kombinované**
  - Kombinace více postupů v jednom vyhledávání.

- **Výhoda**: efektivní využití paměti – až 100% plnění

- **Nevýhoda**: časově náročné vyhledávání – až lineární operační složitost

Tabulky s rozptýlenými položkami

- Používáme, pokud ***N* (rozsah klíče) >> *p* (rozsah tabulky)**
  - Příklad: tabulka pro max. 1000 zaměstnanců identikovatelných podle max. 10 velkých písmen příjmení

$$N = 26^{10} >> p = 1000$$

- Pro určení pozice v tabulce, na kterou máme uložit položku s klíčem k, používáme **rozptylovací funkci** (hash-funkci), která klíči k jednoznačně přiřazuje klíč A_k:

$$A_k = h(k)$$

  - Může se stát, že pro různé položky $k_1 \neq k_2$ platí, že $h(k_1) = h(k_2)$ - vznikají tzv. **synonymické položky** a říkáme, že nastává **kolize**

- **Požadavky na rozptylovací funkci:**
  - Pro každé x je **jednoznačně definovaná** (a v přijatelném čase vyčíslitelná)
  - Vytváří **minimální počet kolizí** (minimum synonym)
  - **Pravděpodobnostní rozdělení $Ak = h(k)$** na intervalu $\langle 0, p\text{-}1 \rangle$ je rovnoměrné, je tedy výhodné použít pseudonáhodné funkce - **randomizační funkce**

- **Realizace $h(x)$** - hash-funkci $i = h(x)$ můžeme úspěšně realizovat například následujícími způsoby:
  - i je částí x
  - i je částí výsledku operace nad x
  - i je zbytkem po dělení rozsahem tabulky p
  - i je zbytkem po dělení N, kde N je nejbližší menší prvočíslo (než rozsah tabulky p)
  - i je dán váhovým součtem částí x:

$$i = \sum_{i=1}^{r} a_i x^i$$

kde a_i jsou váhy jednotlivých částí x^i klíče x

- *Příklad:*

$$h = ((2k + 100p + 1) \bmod N) \bmod p$$

  kde ***p*** je rozsah tabulky a ***k*** je klíč

- Funguje výborně pro tabulky s plněním menším než 70%
- **Operační složitost** vyhledání: jednotková ($f(N) \approx 1$) až logaritmická ($f(N) \approx \log N$)