

IAL – 4. přednáška



Vyhledávací tabulky I.

Obsah přednášky

- Vyhledávací tabulky
 - ADT Vyhledávací tabulka
 - Hodnocení a klasifikace metod
 - Sekvenční vyhledávání

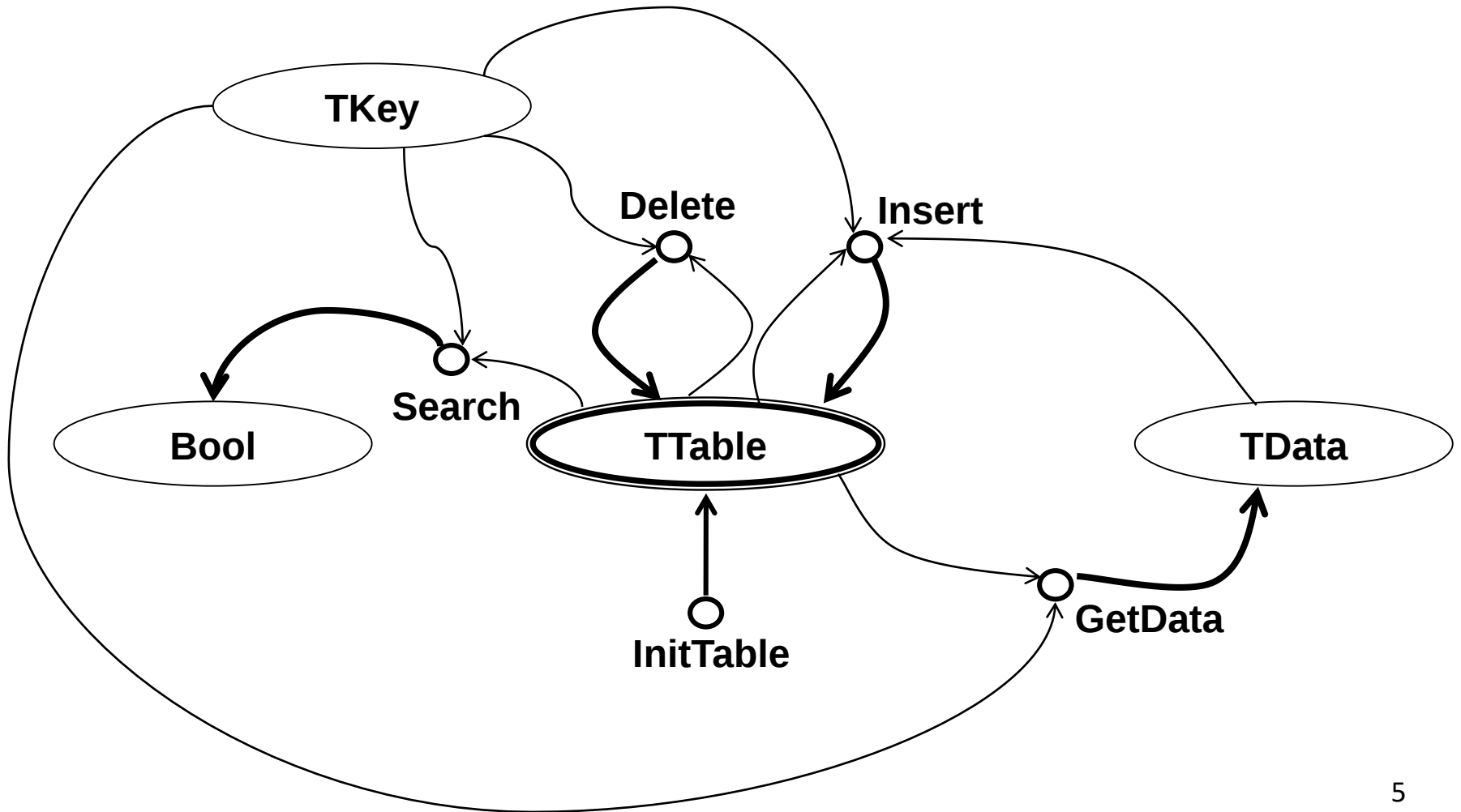
Vyhledávací tabulka

- ❑ Search table, Look-up table
- ❑ Homogenní, obecně dynamická struktura
- ❑ Každá položka má zvláštní složku – **klíč**
- ❑ V tabulce s (ostrým) vyhledáváním je **hodnota klíče jedinečná** (neexistují dvě či více položek se stejnou hodnotou klíče).
- ❑ Tabulka je jako „*kartotéka*“, je to základ databází.

ADT TTable – návrh operací

- ❑ **Inicializace:** InitTable
- ❑ **Vložení prvku:** Insert
- ❑ **Zpřístupnění hodnoty prvku:** GetData
- ❑ **Aktualizace** hodnoty prvku: Insert
- ❑ **Mazání** (rušení) prvku: Delete
- ❑ Operace pro **pohyb** po datové struktuře: -
- ❑ **Predikát:** Search

ADT TTable – diagram signatury



ADT TTable – sémantika operací

- ❑ **InitTable(T)** – inicializuje (vytváří) prázdnou tabulku položek se složkami: klíč **K** typu `TKey` a data **Data** typu `TData`.
- ❑ **Insert(T,K,Data)** – vloží položku se složkami **K** a **Data** do tabulky **T**. Pokud tabulka **T** již obsahuje položku s klíčem **K**, dojde k přepisu datové složky **Data** novou hodnotou – **aktualizační sémantika** operace Insert.
- ❑ **Delete(T,K)** – zruší prvek s klíčem **K**. Pokud prvek neexistuje, je bez účinku (prázdná operace).
- ❑ **Search(T,K)** – predikát, který vrací hodnotu *true*, pokud se v tabulce **T** nachází položka s klíčem **K**, v opačném případě vrací hodnotu *false*.

ADT TTable – sémantika operací

- **GetData(T,K)** – operace vrátí hodnotu datové složky položky s klíčem **K**. Pokud položka s klíčem **K** v tabulce **T** neexistuje, dojde k chybě. Operaci je potřeba vždy ošetřit použitím predikátu Search(T,K):

```
if Search (T,K) :  
    El ← GetData (T,K)
```

- *Pozn.:* Implementace operací pro ADT TTable závisí na zvoleném způsobu organizace dat.

Základní pojmy a klasifikace

- Přístupová doba (angl. *Access Time*)
- Doba vyhledání
 - minimální
 - maximální
 - průměrná/střední
 - při úspěšném vyhledání
 - při neúspěšném vyhledání
- Vyhledávání v datové struktuře
 - s přímým přístupem
 - se sekvenčním přístupem

Metody implementace tabulky (1/2)

- Sekvenční vyhledávání v **neseřazeném poli**
- Sekvenční vyhledávání v neseřazeném poli se zářátkou
- **Sekvenční** vyhledávání v **seřazeném poli**
- Sekvenční vyhledávání v poli seřazeném podle pravděpodobnosti vyhledání klíče
- Sekvenční vyhledávání v poli s adaptivním uspořádáním podle četnosti vyhledání

Metody implementace tabulky (2/2)

- ❑ **Binární vyhledávání v seřazeném poli**
 - normální binární vyhledávání
 - Dijkstrova varianta binárního vyhledávání
- ❑ Binární vyhledávací stromy (BVS)
- ❑ AVL stromy
- ❑ Stromy s více klíči ve vrcholech
- ❑ Tabulky s rozptýlenými položkami
(angl. *Hash tables*) – TRP

Typy vyhledávacích tabulek

□ Statické tabulky:

- Množina prvků je fixní.
- Důraz na rychlé vyhledávání.

□ Dynamické tabulky:

- Prvky lze přidávat nebo i odebírat.
- Je třeba zvážit časovou složitost i těchto operací.
- Dynamika 1. stupně – prvky pouze vkládáme.
- Dynamika 2. stupně – vkládání i rušení prvků.

Sekvenční vyhledávání

□ Dohoda:

- Pro typ klíče budeme používat nejčastěji identifikátor `TKey`,
- pro název klíčové složky položky tabulky identifikátor `key`,
- a pro hodnotu vyhledávaného klíče identifikátor `k`.

□ Nad typem `TKey` rozlišujeme dva typy relací:

- Relace rovnosti
- Relace uspořádání

□ Pro sekvenční vyhledávání stačí, aby nad typem `TKey` byla definována relace rovnosti.

Vyhledávací algoritmus

□ Základní struktura vyhledávacího algoritmu:

```
found ← false
```

```
while (not found and <množina prvků není vyčerpána>) :  
    <prozkoumej další prvek, a je-li to hledaný,  
    proved' found ← true>
```

```
Search ← found
```

Vyhledávací algoritmus

- Pozor na ošetření konce cyklu – možné řešení:

```
found ← false
i ← 0
while not found and (i < max):
    if k = array[i].key:
        found ← true
    else:
        i ← i+1
Search ← found
```

Vyhledávací algoritmus

- Pozor na ošetření konce cyklu – **nevhodné řešení:**

```
i ← 0  
while (k ≠ array[i].key) and (i < max) :  
    i ← i+1  
Search ← k = array[i].key
```

- Pokud hledaný klíč v poli není, dojde k **přístupu na adresu za hranicí pole!**
 - Toto řešení lze použít pouze pokud prohodíme použité podmínky a použijeme zkratové (neúplné) vyhodnocování booleovských výrazů.

Vyhledávací algoritmus

□ Zkratové vyhodnocování booleovských výrazů:

■ **B1 and B2 and B3 and ... and BN**

⇒ je-li B1 *false*, vše je *false*

■ **B1 or B2 or B3 or ... or BN**

⇒ je-li B1 *true*, vše je *true*

□ Vyhledávací algoritmus pak může mít tvar:

```
i ← 0
```

```
// zkratové vyhodnocení Booleovského výrazu
```

```
while (i < max) and (k ≠ array[i].key):
```

```
    i ← i+1
```

```
Search ← i < max
```

Vyhledávací algoritmus

- Ošetření konce cyklu v seznamu – možné řešení:

```
found ← false
while not found and ptr ≠ NULL:
    if k = ptr->key:
        found ← true
    else:
        ptr ← ptr->nextPtr
```

- Využití neúplného vyhodnocení logického výrazu:

```
... while (ptr ≠ NULL) and (k ≠ ptr->key):
    ...
```

Vyhledávací algoritmus

- ❑ Špatné ošetření konce cyklu:

```
ptr ← l.first  
while (k ≠ ptr->key) and (ptr ≠ NULL):  
    // chybná reference  
    ptr ← ptr->nextPtr
```

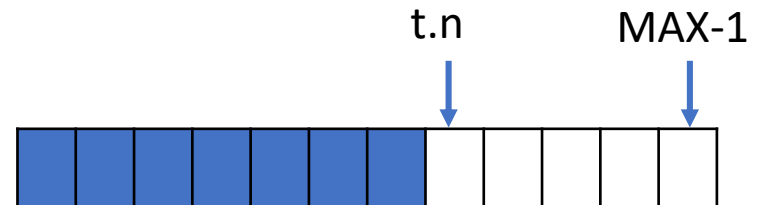
Sekvenční vyhledávání – implementace

- Definujeme následující datové typy:

```
#define MAX ...
```

```
typedef struct telem
{
    TKey key;
    TData data;
}TElem;
```

// typ položky tabulky



```
typedef struct ttable
{
    TElem array[MAX]; // typ tabulka implementovaná polem
    int n;             // pole tabulky
}TTable;              // aktuální počet prvků v tabulce
```

Operace Search

```
bool function Search (TTable t, TKey k)
    found ← false
    i ← 0
    while not found and (i < t.n):
        if k = t.array[i].key:
            found ← true
        else:
            i ← i+1
    return (found)
```

Varianta Search pro vkládání

- Varianta operace Search, která vrací polohu (index) hledaného prvku:

```
(bool, int) function SearchInd (TTable t, TKey k)
    found ← false
    i ← 0
    while not found and (i < t.n):
        if k = t.array[i].key:
            found ← true
        else:
            i ← i+1
    where ← i           // pro not found je i nedefinováno
    return (found, where)
```

Operace Insert

```
bool function Insert (TTable t, TElem el)
    overflow ← false           // příznak plné tabulky
    found, where ← SearchInd(t, el.key)
    if found:
        t.array[where] ← el    // přepsání staré položky
    else:
        if t.n < MAX:           // je místo - vkládáme
            t.array[t.n] ← el
            t.n ← t.n+1
        else:
            overflow ← true    // nelze vložit - přetečení
    return (not overflow)
```

Operace Delete

```
procedure Delete (TTable t, TKey k)
  found, where  $\leftarrow$  SearchInd(t,k)
  if found:                                // rušený je přepsán posledním
    t.array[where]  $\leftarrow$  t.array[t.n-1]
    t.n  $\leftarrow$  t.n-1
```

- Operaci **Delete** lze také implementovat **zaslepením**:
 - Klíč rušené položky se přepíše hodnotou, která se nikdy nebude vyhledávat.
 - Snižuje aktivní kapacitu tabulky!

Hodnocení sekvenčního vyhledávání

- Minimální čas úspěšného vyhledání: 1
- Maximální čas úspěšného vyhledání: n
- Průměrný čas úspěšného vyhledání: $n/2$
- Čas neúspěšného vyhledání: n
- Nejrychleji jsou vyhledány položky, které jsou na začátku tabulky.

Sekvenční vyhledávání se zarážkou

□ **Zarážka** (*sentinel, guard, stop-point*):

- Dovoluje vynechat test na konec pole.
- Sníží efektivní kapacitu tabulky o jednu položku.
- Vynecháním testu na konec se algoritmus zrychlí.

```
bool function SearchG (TTable t, TKey k)
    i ← 0
    t.array[t.n].key ← k      // vložení zarážky
    while k ≠ t.array[i].key:
        i ← i+1
    return (i ≠ t.n)
    // když našel až zarážku, tak vlastně nenašel ...
```

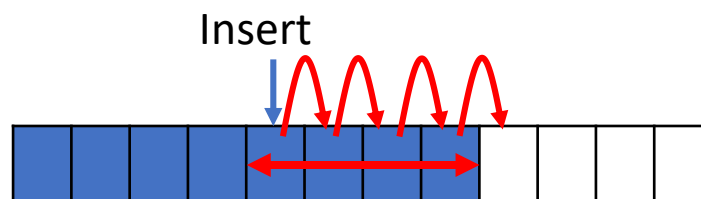
Sekvenční vyhledávání v seřazeném poli

- Pole je **seřazeno** podle velikosti klíče:
 - Nad typem klíč musí být definována relace uspořádání.
- Operace **Search**:
 - Skončí neúspěšně, jakmile narazí na položku s klíčem, který je větší než hledaný klíč.
 - **Urychlí se pouze neúspěšné vyhledávání!**
- Operace **Insert** a **Delete**:
 - Musí **zachovat uspořádání pole!**
 - Vyžadují proto **posuny segmentů pole!**

Sekvenční vyhledávání v seřazeném poli

□ Operace **Insert**:

- Musí najít správné místo pro vložení prvku.
- **Segment pole** od nalezeného místa se musí **posunout** o jednu pozici vpravo.



□ Operace **Delete**:

- **Segment pole** vpravo od mazaného prvku se musí **posunout** o jednu pozici vlevo – přepíše rušený prvek.

Posuny segmentů pole

- Posun segmentu doprava se provede cyklem zprava (od konce pole) – posun segmentu $[low..(high-1)]$ o jednu pozici doprava:

```
for  $i \leftarrow (high, low+1)^{-1}$ :  
     $t.array[i] \leftarrow t.array[i-1]$ 
```

Seřazení pole – četnost vyhledávání

- V **praxi** jsou často některé položky vyhledávány **mnohem častěji než ostatní** – při použití sekvenčního vyhledávání se vyplatí tyto položky umístit na **začátek pole**, aby byly **nalezeny rychleji**.
- **Seřazení pole podle četnosti** vyhledávání:
 - **Jednou za čas** – lze realizovat pomocí počítadla, které se aktualizuje po každém přístupu k položce.
 - **Průběžně** – **adaptivní rekonfigurace** podle četnosti vyhledávání – při každém přístupu k položce se položka vymění se svým levým sousedem (pokud existuje).

Seřazení pole – četnost vyhledávání

- Při **adaptivní rekonfiguraci** je součástí operace **Search** při úspěšném vyhledání příkaz:

```
if where > 0:
```

```
    t.array[where] ↔ t.array[where-1]
```

- *Pozn.:* zápis $A \leftrightarrow B$ označuje operaci výměny hodnot, kterou je obvykle potřeba realizovat trojicí příkazů a pomocnou proměnnou. Tuto operaci je v rámci IAL možné použít vždy pro usnadnění zápisu.