IAL – 3. přednáška

Abstraktní datové typy II.

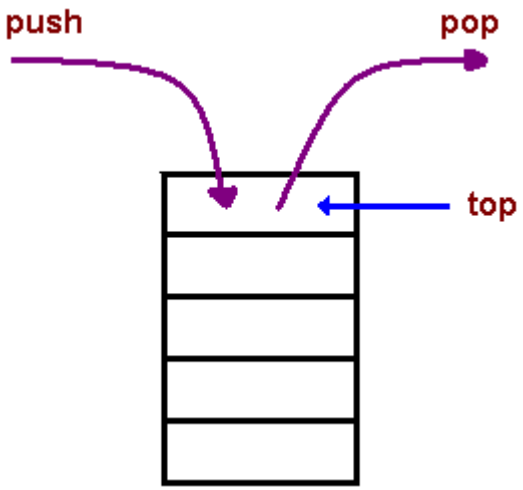
1. a 2. října 2024

Obsah přednášky

- **ADT zásobník**
  - Implementace zásobníku polem a seznamem
  - Vyčíslování aritmetických výrazů v postfixové notaci se zásobníkem
  - Převod z infixové do postfixové notace se zásobníkem
- ADT fronta
  - Implementace fronty polem a seznamem
- ADT vyhledávací tabulka
- Poznámky k implementaci vícerozměrných polí
  - Prostorově úsporné metody implementace některých polí

Zásobník – Stack

- Homogenní, dynamický, lineární datový typ
- **LIFO:** Last In – First Out
- Použití:
  - Reverze pořadí prvků
  - Konstrukce (rekurzivních) funkcí
  - Vyčíslování aritmetických výrazů
  - Převod infixové notace na postfixovou či prefixovou
  - Operace zpět/znovu (Undo/Redo) v editorech
  - Prohledávání s návratem (angl. Backtracking)

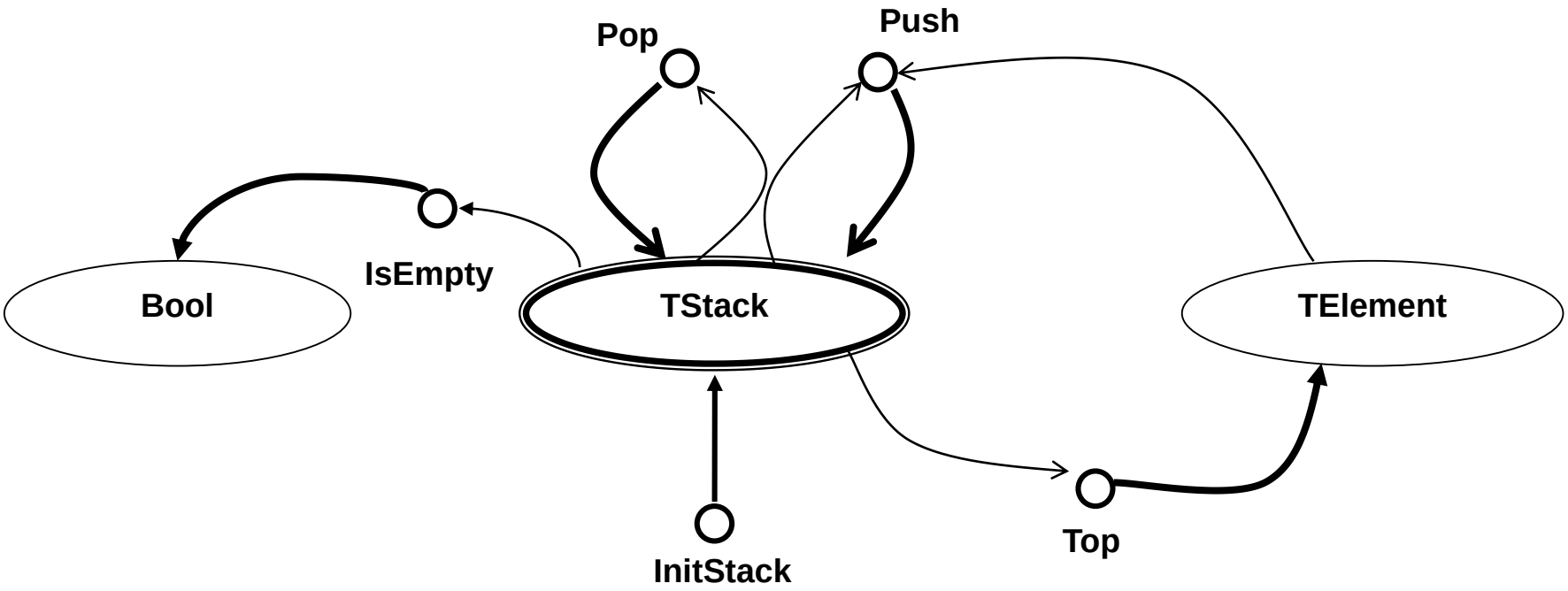


ADT TStack – návrh operací

- **Inicializace**: InitStack
- **Vložení** prvku: Push
- **Zpřístupnění** hodnoty prvku: Top
- **Aktualizace** hodnoty prvku: -
- **Mazání** (rušení) prvku: Pop
- Operace pro **pohyb** po datové struktuře: -
- **Predikát**: IsEmpty

- *Pozn.:* Často je zásobník implementován tak, že operace Top vrací hodnotu prvku na vrcholu zásobníku a zároveň tuto hodnotu odstraňuje. My chceme, aby každá operace dělala jen jednu věc, proto definujeme samostatné operace Top a Pop.

ADT TStack – diagram signatury



ADT TStack – sémantika operací

- **InitStack(S)** – vytvoří prázdný zásobník.
- **Push(S,El)** – vloží prvek **El** na vrchol zásobníku.
- **Pop(S)** – zruší (odstraní) prvek z vrcholu zásobníku (při prázdném zásobníku bez účinku).
- **IsEmpty(S)** – predikát, který vrací hodnotu *true* je-li zásobník prázdný, v ostatních případech vrací hodnotu *false*.
- **Top(S)** – vrací hodnotu prvku na vrcholu zásobníku. Obsah zásobníku ponechá beze změny. Pro prázdný zásobník dojde k **chybě**. Operaci Top je potřeba vždy ošetřit testem na neprázdnost zásobníku:

```
if not IsEmpty(S):
    El ← Top(S)
    ...
```

Axiomy sémantiky ADT TStack

1. Pop(InitStack(S)) = InitStack(S)
2. Pop(Push(S,El)) = S
3. Top(InitStack(S)) = error
4. Top(Push(S,El)) = El
5. IsEmpty(InitStack(S)) = true
6. IsEmpty(Push(S,El)) = false

ADT TStack – implementace

- Zásobník je lineární struktura – může být implementován polem nebo spojovým seznamem.
- **Implementace polem:**
  - Datová struktura bude *pseudodynamická* – dynamická tak dlouho, dokud nevyčerpá alokovaný prostor.
  - Je vhodné doplnit množinu operací o predikát, který bude signalizovat naplnění alokovaného prostoru a tím také nemožnost vložení dalšího prvku do zásobníku.
  - **IsFull(S)** – predikát, který bude vracet hodnotu *true* v případě plného zásobníku, v ostatních případech bude vracet hodnotu *false*.
- *Pozn.*: V případě využití nafukovacího pole toto není potřeba řešit. Musíme si ale být vědomi vlastností tohoto pole.

ADT TAStack – implementace polem

```
#define SMAX 1000        // maximální kapacita zásobníku

typedef struct tastack
{
   TData dataArray[SMAX];           // pole pro zásobník
   int top;                  // index prvního volného prvku
} TAStack;



void A_InitStack (TAStack *s)
{
  s->top = 0;
}
```

ADT TAStack – implementace polem

```
void A_Push (TAStack *s, TData d)
{ /* Je-li zaručena kontrola !A_IsFull(s),
     lze test na plnost zásobníku vynechat. */
  if (s->top < SMAX) {
    s->dataArray[s->top] = d;
    s->top++;
  }
}

void A_Pop (TAStack *s)
{// Operace Pop nevrací hodnotu.
  if (s->top > 0) {
    s->top--;
  }
}
```

ADT TAStack – implementace polem

```
TData A_Top (TAStack *s)
{ /* Podmínkou vyvolání této funkce je test na
neprázdnost zásobníku. Je-li zásobník prázdný, funkce
nevrátí správnou hodnotu. */
  return (s->dataArray[(s->top)-1]);
}

bool A_IsEmpty (TAStack *s)
{
 return (s->top == 0);
}

bool A_IsFull (TAStack *s)
{
  return (s->top == SMAX);
}
```

ADT TStack – implementace seznamem

```
typedef struct tselem
{
      TData data;
      struct tselem *next;
}TSElem

typedef struct tstack
{
      TSElem *top;
}TStack
```

ADT TStack – implementace seznamem

```
void InitStack (TStack *s)
{
  s->top = NULL;
}


void Push (TStack *s, TData d)
{
  TSElem *newElemPtr = (TSElem *) malloc(sizeof(TSElem));
                // zkontrolovat úspěšnost operace malloc!
  newElemPtr->data = d;
  newElemPtr->next = s->top;
  s->top = newElemPtr;
}
```

ADT TStack – implementace seznamem

```
void Pop (TStack *s)
{
  TSElem *elemPtr;
  if (s->top != NULL) {
    elemPtr = s->top;
    s->top = s->top->next;
    free(elemPtr);
  }
}
```

ADT TStack – implementace seznamem

```
TData Top (TStack *s)
/* V případě prázdného zásobníku dojde k chybě.
Otestovat před voláním funkce! */
{
  return (s->top->data);
}


bool IsEmpty (TStack *s)
{
  return (s->top == NULL);
}
```

K procvičení:

- Bylo by možné implementovat ADT zásobník výhradně pomocí operací pro ADT jednosměrně vázaný seznam?
- Pokud ano, jak by implementace jednotlivých operací vypadala?
- Podobně se lze zamyslet i nad ADT fronta…

Zápis matematických výrazů

- Pro zápis matematických výrazů lze použít různé notace:
  - Infixová ⇨ a + b
  - Prefixová (polská) ⇨ + a b
    - připomíná zápis funkce … `int ADD (a,b:int)`
    - Jan Łukasiewicz, 1878-1956
  - Postfixová (obrácená polská) ⇨ a b +

    x + y = ⇨ x y + =

    (a+b)*(c-d)/(e+f)*(g-h)= ⇨ ab+cd-*ef+/gh-*=
    - Charles Leonard Hamblin, 1922-1985

Postfixová notace

- Výhody:
  - Neobsahuje závorky
  - **Snadné vyčíslení** – operace se provádějí v pořadí operátorů v řetězci

    Výraz: a b + c d - * e f + / g h - * =
    se zpracuje jako by měl tvar:
    (((a b +) (c d -) *) (e f +) /) (g h -) * =

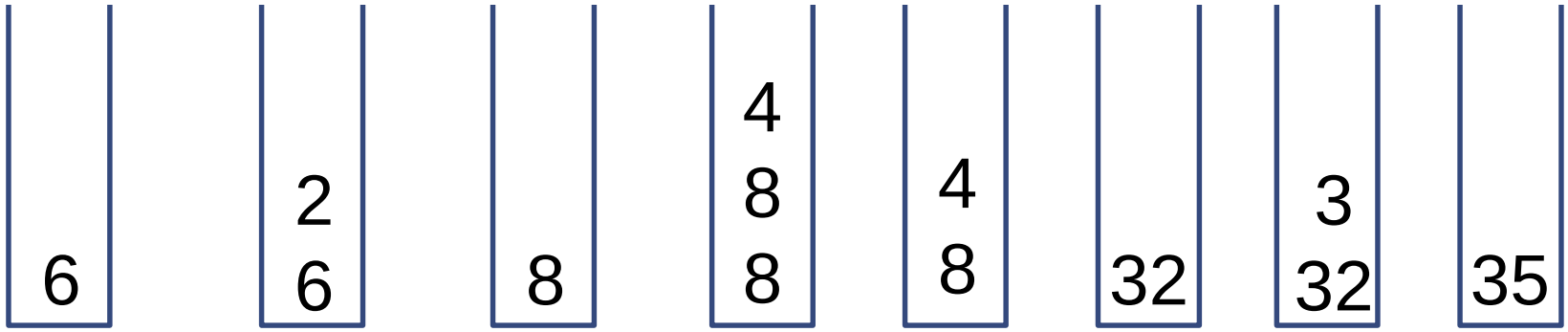
Vyčíslení postfixového výrazu

- **Algoritmus:**
  1. Zpracovávej řetězec zleva doprava.
  2. Je-li zpracovávaným prvkem operand, vlož ho do zásobníku.
  3. Je-li zpracovávaným prvkem operátor, vyjmi ze zásobníků tolik operandů, kolika-adický je operátor (pro dyadické operátory dva operandy), proveď danou operaci a výsledek ulož na vrchol zásobníku.
  4. Je-li zpracovávaným prvkem omezovač ‘=‘, je výsledek na vrcholu zásobníku.

Příklad: Vyčíslení postfixu

(6+2)*(8-4)+3=

6 2 + 8 4 - * 3 + =



K procvičení:

- Předpokládejte, že je definován ADT `TStack` čísel `int` a deklarována globální proměnná `S` tohoto typu, nad kterou smíte aplikovat všechny operace nad `TStack`. Je dán řetězec znaků, obsahující číslice, operátory `+`, `-`, `*` a `/`, které představují operace nad typem `int` a ukončovací znak (omezovač) `=`, kterým je řetězec zakončen. Číslice představují jednomístná celá čísla. Předpokládejte, že řetězec představuje syntakticky správný aritmetický výraz.
- **Napište funkci**, která má na vstupu řetězec s postfixovým výrazem a která vrátí hodnotu vyčísleného výrazu.

Převod infixové notace na postfixovou

1. Zpracovávej vstupní řetězec položku po položce zleva doprava a vytvářej postupně výstupní řetězec.
2. Je-li zpracovávanou položkou **operand**, přidej ho na konec vznikajícího výstupního řetězce.
3. Je-li zpracovávanou položkou **levá závorka**, vlož ji na vrchol zásobníku.

Převod infixové notace na postfixovou

4. Je-li zpracovávanou položkou **operátor**, pak ho na vrchol zásobníku vlož v případě, že:
   - zásobník je prázdný
   - na vrcholu zásobníku je levá závorka
   - na vrcholu zásobníku je operátor s nižší prioritou

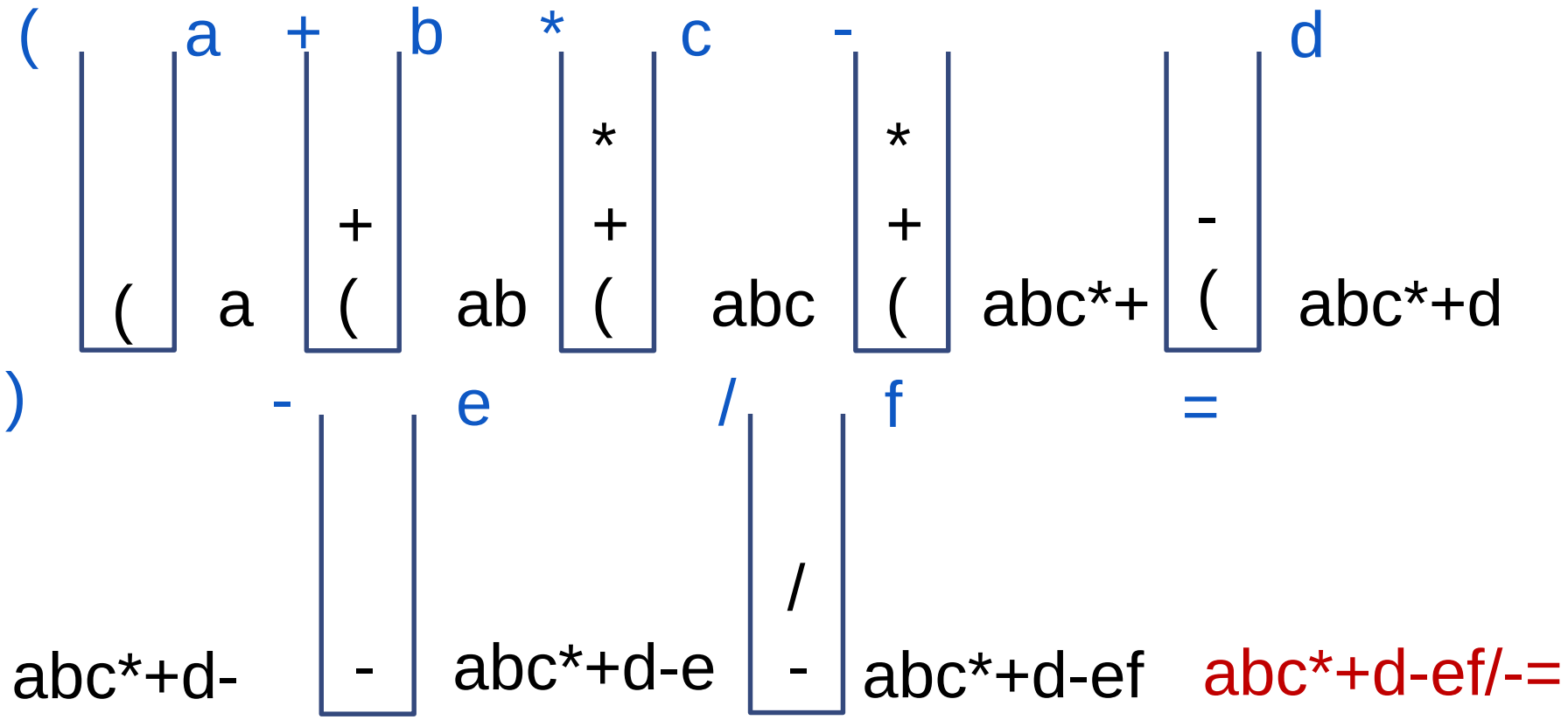
   Je-li na vrcholu zásobníku operátor s **vyšší nebo shodnou** prioritou, odstraň ho, vlož ho na konec výstupního řetězce a opakuj krok 4, až se ti podaří operátor vložit na vrchol.

Převod infixové notace na postfixovou

5. Je-li zpracovávanou položkou pravá závorka, odebírej z vrcholu položky a dávej je na konec výstupního řetězce, až narazíš na levou závorku. Levou závorku odstraň ze zásobníku. Tím je pár závorek zpracován.
6. Je-li zpracovávanou položkou omezovač =, pak postupně odstraňuj prvky z vrcholu zásobníku a přidávej je na konec řetězce, až zásobník zcela vyprázdníš, a na konec přidej rovnítko.

Příklad: převod infix → postfix

(a+b*c-d)-e/f=

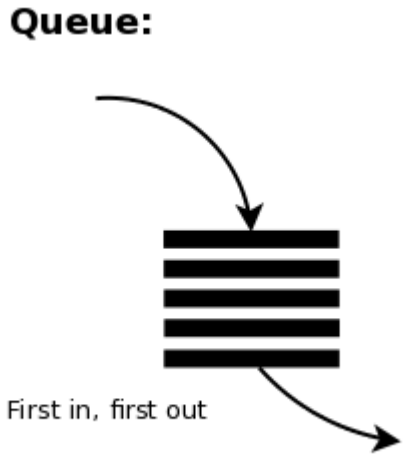


Obsah přednášky

- ADT zásobník
  - Implementace zásobníku polem a seznamem
  - Vyčíslování aritmetických výrazů v postfixové notaci se zásobníkem
  - Převod z infixové do postfixové notace se zásobníkem
- **ADT fronta**
  - Implementace fronty polem a seznamem
- ADT vyhledávací tabulka
- Poznámky k implementaci vícerozměrných polí
  - Prostorově úsporné metody implementace některých polí

Fronta – Queue

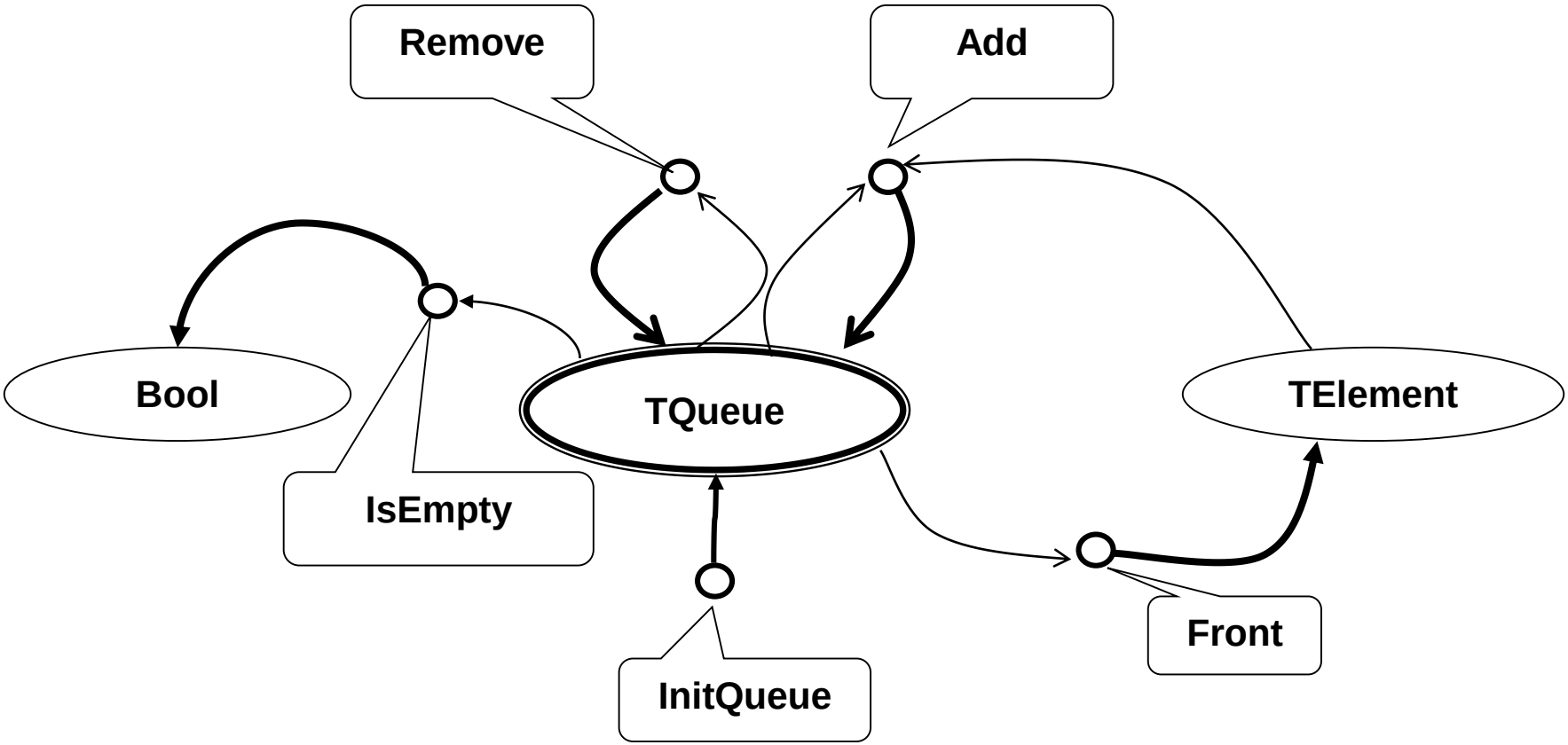
- Dynamická, homogenní a lineární struktura.
- **FIFO:** First In – First Out
- Na jedné straně přidává (konec fronty) a na druhé čte a odebírá – obsluhuje (začátek fronty).
- Teorie front (teorie hromadné obsluhy) – historie s vodovodními kohoutky



ADT TQueue – návrh operací

- **Inicializace**: InitQueue
- **Vložení** prvku: Add
- **Zpřístupnění** hodnoty prvku: Front
- **Aktualizace** hodnoty prvku: -
- **Mazání** (rušení) prvku: Remove
- Operace pro **pohyb** po datové struktuře: -
- **Predikát**: IsEmpty

ADT TQueue – diagram signatury



ADT TQueue – sémantika operací

- **InitQueue(Q)** – inicializace (prázdné) fronty
- **Add(Q,El)** – vloží prvek **El** na konec fronty.
- **IsEmpty(Q)** – predikát, který vrací hodnotu *true* je-li fronta prázdná, v ostatních případech vrací hodnotu *false*.
- **Front(Q)** – funkce, která vrací hodnotu prvku na začátku fronty. Obsah fronty se nemění. Pro prázdnou frontu dojde k chybě. Operaci je vždy potřeba ošetřit testem na neprázdnost:

```
if not IsEmpty(Q):
    El ← Front(Q)
    ...
```

- **Remove(Q)** – odstraní (zruší) prvek ze začátku fronty (pro prázdnou frontu je bez účinku).

ADT TQueue – axiomy sémantiky

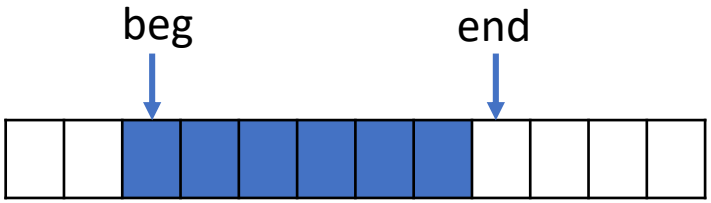
1. IsEmpty(InitQueue(Q)) = true
2. IsEmpty(Add(Q,El)) = false
3. Front(InitQueue(Q)) = error
4. Front(Add(InitQueue(Q),El)) = El
5. Front(Add(Add(InitQueue(Q),ElA),ElB)) = Front(Add(InitQueue(Q),ElA))
6. Remove(InitQueue(Q)) = InitQueue(Q)
7. Remove(Add(InitQueue(Q),El)) = InitQueue(Q)
8. Remove(Add(Add(InitQueue(Q),ElA),ElB)) = Add(Remove(Add(InitQueue(Q),ElA)),ElB)

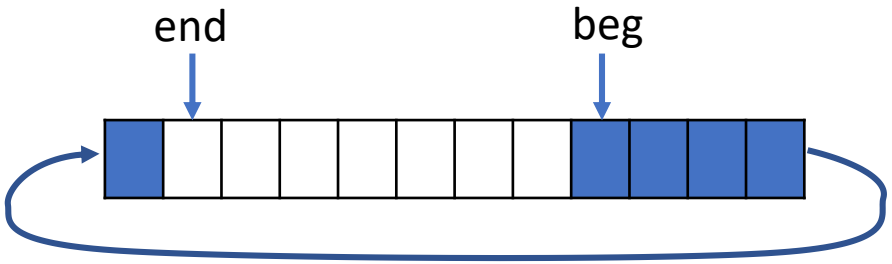
ADT TQueue – implementace

- Frontu lze opět implementovat polem nebo spojovým seznamem
- **Implementace polem:**
  - Struktura bude opět *pseudodynamická*, je vhodné doplnit predikát IsFull(Q).
  - Potřebujeme „posouvat“ začátek i konec fronty – posuny segmentů pole jsou pomalé, použijeme kruhově svázané pole.
  - Pomocí dvojice indexů hlídáme pozici začátku fronty (prvního prvku) a pozici pro vložení prvku na konec fronty (první volné místo za posledním prvkem).
  - **Efektivní kapacita** fronty bude o 1 menší než velikost pole (abychom rozlišili plnou a prázdnou frontu).

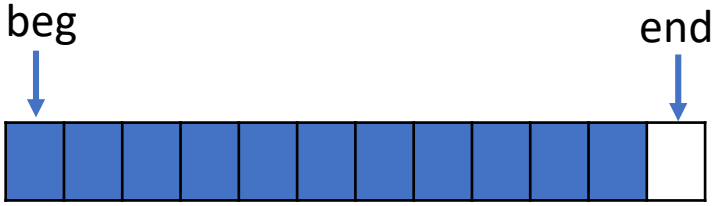
ADT TAQueue – implementace polem

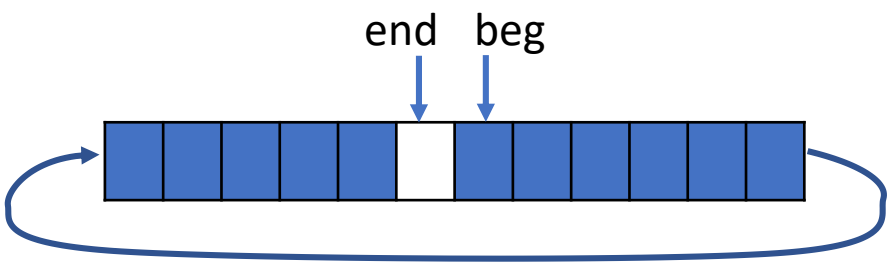
Běžný stav fronty (beg – první prvek):



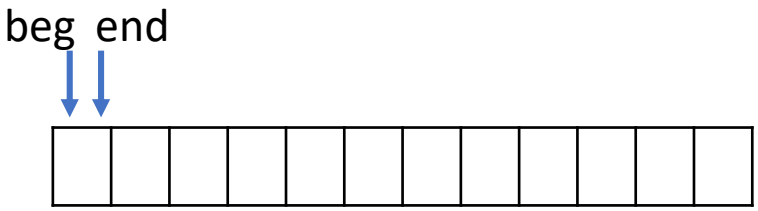


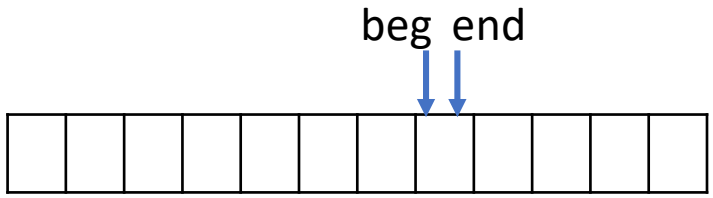
Plná fronta:





Prázdná fronta:





ADT TAQueue – implementace polem

```
#define QMAX 1000   // fronta má kapacitu QMAX-1 prvků

typedef struct taqueue
{
    TData dataArray[QMAX];
    int beg, end;
} TAQueue;


void A_InitQueue (TAQueue *q)
{
  q->beg = 0;
  q->end = 0;
}
```

ADT TAQueue – implementace polem

```
TData A_Front (TAQueue *q)
{ /* Ověřit neprázdnost fronty před voláním funkce */
  return (q->dataArray[q->beg]);
}

bool A_IsEmpty (TAQueue *q)
{
  return (q->beg == q->end);
}
```

Soubor lze doplnit o predikát plnosti fronty takto:

```
bool A_IsFull (TAQueue *q)
{
   return (((q->beg -1) == q->end) ||
           ((q->beg == 0) && (q->end == QMAX-1)));
}
```

ADT TAQueue – implementace polem

```
void A_Add (TAQueue *q, TData d)
{
  if (!A_IsFull(q))
    q->dataArray[q->end] = d;
    q->end++;
    if (q->end == QMAX){ // ošetření kruhovosti pole
      q->end = 0;
    }
  } // if q není plná
}
```

ADT TAQueue – implementace polem

```
void A_Remove (TAQueue *q)
{
  if (!A_IsEmpty(q)){
    q->beg++;
    if (q->beg == QMAX){ // ošetření kruhovosti pole
      q->beg = 0;
    }
  }
}
```

Kruhovost pole – operace modulo

- ***Pozn.:*** Kruhovost pole lze zajistit také pomocí operace **modulo**:

  beg = (beg + 1) % QMAX

- Například když je QMAX = 100 (pole má indexy 0 až QMAX-1), pak:
  - když staré beg = 98, pak po inkrementaci je: *beg = (98+1) % 100 = 99*
  - když staré beg = 99, pak po inkrementaci je: *beg = (99+1) % 100 = 0*

- Pokud by pole začínalo indexem 1, je nutné tuto jedničku přičíst. Pak lze inkrementaci ukazatele zajistit příkazem:

  beg = beg % QMAX +1

- Například když je QMAX = 100 (pole má indexy 1 až QMAX), pak:
  - když staré *beg = 99,* pak po inkrementaci je: *beg = 99 % 100 +1 = 100*
  - když staré *beg = 100,* pak po inkrementaci je: *beg = 100 % 100 +1 = 1*

ADT TQueue – implementace seznamem

```
typedef struct tqelem
{
  TData data;
  struct tqelem *next;
} TQElem;

typedef struct tqueue
{
  TQElem *beg;
  TQElem *end;
}TQueue;
```

ADT TQueue – implementace seznamem

```
void InitQueue (TQueue *q)
{
   q->beq = NULL;
   q->end = NULL;
}

bool IsEmpty (TQueue *q)
{
   return (q->beg == NULL);
}
```

ADT TQueue – implementace seznamem

```
void Add (TQueue *q, TData d)
{
  TQElem *newElemPtr = (TQElem *)malloc(sizeof(TQElem));
            //zkontrolovat úspěšnost operace malloc!
  newElemPtr->data = d;          //naplnění nového prvku
  newElemPtr->next = NULL;       //ukončení nového prvku
  if (q->beg == NULL){  // fronta je prázdná,
    q->beg = newElemPtr;//vlož nový jako první a jediný
  }
  else {//obsahuje nejméně jeden prvek, přidej na konec
    q->end->next = newElemPtr;
  }
  q->end = newElemPtr;           // korekce konce fronty
}
```

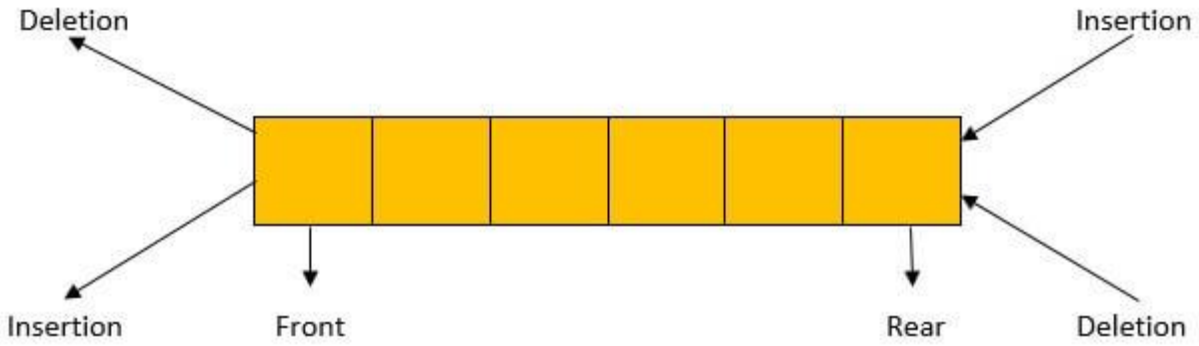
ADT TQueue – implementace seznamem

```
TData Front (TQueue *q)
{ /*Ověřit neprázdnost fronty před voláním funkce*/
  return (q->beg->data);
}

void Remove (TQueue *q)
{
  if (q->beg != NULL){           //Fronta je neprázdná
    TQElem *elemPtr = q->beg;
    if (q->beg == q->end){
      q->end = NULL;     //Zrušil se poslední a jediný
    }
    q->beg = q->beg->next;
    free(elemPtr);
  }
}
```

Oboustranná fronta – Deque

- Double Ended Queue – fronta u níž lze přidávat/odebírat prvky na obou koncích
- Na rozdíl od seznamu nelze touto frontou nějak procházet a manipulovat s ní někde uvnitř.
- Další označení: oboustranný zásobník, head-tail linked list
- Příklad použití: při zpracování (plánování) procesů – při rozdvojení procesu může být jedna větev vložena na „začátek“ fronty



Prioritní fronta

- Prvkům fronty je navíc přiřazena **priorita**.
- Prvky s vyšší prioritu přeskakují prvky s nižší prioritou a jsou obsluhovány dříve než prvky s nižší prioritou.
- Jako první opouští frontu nejstarší prvek s nejvyšší prioritou.
- Operace:
  - **Inicializace**: InitPriorityQueue
  - **Vložení** prvku: AddWithPriority
  - **Zpřístupnění** hodnoty prvku: GetHighestPriorityElement
  - **Mazání** (rušení/odebrání) prvku: RemoveHighestPriorityElement
  - **Predikát**: IsEmpty

- *Pozn.:* Pro malé množství priorit ji lze realizovat pomocí oddělených front pro jednotlivé priority.

Prioritní fronta

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

Obsah přednášky

- ADT zásobník
  - Implementace zásobníku polem a seznamem
  - Vyčíslování aritmetických výrazů v postfixové notaci se zásobníkem
  - Převod z infixové do postfixové notace se zásobníkem
- ADT fronta
  - Implementace fronty polem a seznamem
- **ADT vyhledávací tabulka**
- Poznámky k implementaci vícerozměrných polí
  - Prostorově úsporné metody implementace některých polí

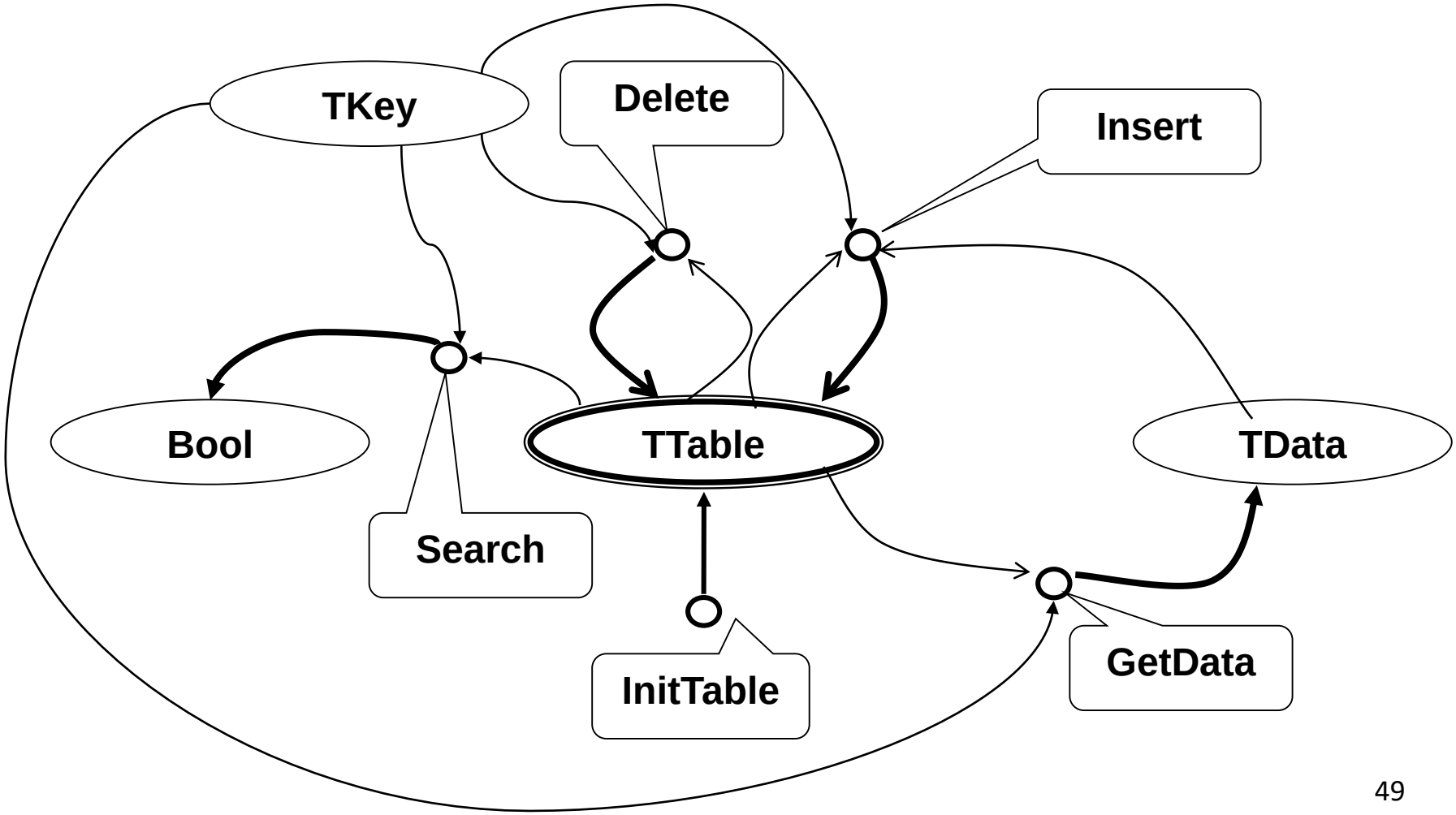
Vyhledávací tabulka

- 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 vrací 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. Tato problematika bude vysvětlena v následujících přednáškách.

Obsah přednášky

- ADT zásobník
  - Implementace zásobníku polem a seznamem
  - Vyčíslování aritmetických výrazů v postfixové notaci se zásobníkem
  - Převod z infixové do postfixové notace se zásobníkem
- ADT fronta
  - Implementace fronty polem a seznamem
- ADT vyhledávací tabulka
- **Poznámky k implementaci vícerozměrných polí**
  - Prostorově úsporné metody implementace některých polí

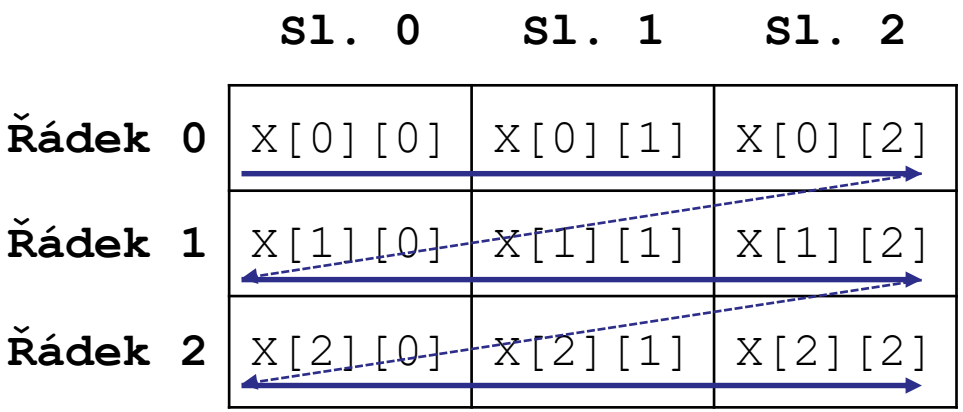
Vícerozměrná pole

- N-rozměrné pole můžeme chápat jako jednorozměrné pole položek, jimiž jsou (N-1) rozměrná pole.
- Průchody:
  - Po řádcích – rychlost změny indexu se snižuje zprava doleva (nejrychleji se mění nejpravější index).
    Příklad: matice – pro každý řádek se postupně mění hodnoty sloupcového (pravého) indexu.
  - Po sloupcích – nejrychleji se mění nejlevější index.

Vícerozměrná pole

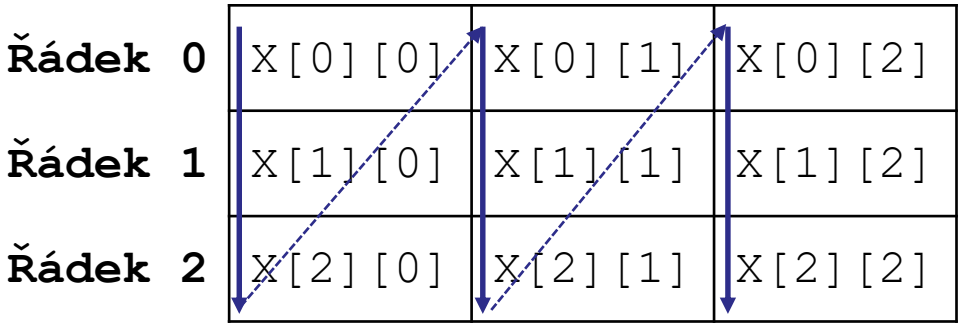
- Po řádcích:
  - rychlost změny indexu se snižuje zprava doleva

```
for i ← (0, rows-1):
  for j ← (0, cols-1):
    processItem(X[i][j])
```



- Po sloupcích:

```
for j ← (0, cols-1):
  for i ← (0, rows-1):
    processItem(X[i][j])
```



Mapovací funkce

- Převádí n-tici indexů prvku n-dimenzionálního pole na jeden index jednorozměrného pole.
- Závisí na tom, jak je n-dimenzionální pole uloženo v paměti (po řádcích nebo po sloupcích).
- Zajišťuje zpřístupnění prvku pole (realizuje funkci selektoru).
- Hodnota mapovací funkce se musí vyčíslovat při každé referenci (odkazu) na indexovanou proměnnou. Vyčíslení může být zejména u vícedimenzionálních polí časově náročné.

Mapovací funkce

- Mějme k-rozměrné pole A, které je v paměti uloženo **po sloupcích**:

  A: array [low_1..$high_1$, low_2..$high_2$, … , low_k..$high_k$] of TElement;

- Potom prvek **A[j_1, j_2, … , j_k]**, bude zobrazen (mapován) do jednorozměrného pole **B** na index určený mapovací funkcí:

$$A[j_1, j_2, \dots, j_k] \rightarrow B\left[low_{1B} + \sum_{m=1}^{k} (j_m - low_m) * D_m\right]$$

kde

$D_1 = 1$

$D_m = (high_{m-1} - low_{m-1} + 1) * D_{m-1}$

Trojúhelníková matice

- Běžné uložení celé matice *N*N* položek.
- Prostorově efektivní uložení – prvky uložíme do 1-rozměrného pole za sebe (po řádcích):
  - Potřebujeme prostor pouze pro *(N/2)*(N+1)* položek.
  - Pro mapování po řádcích se použije funkce

$$A[i,j] \rightarrow B[i * (i+1)\ div\ 2 + j]$$

$a_{0,0}$				
$a_{1,0}$	$a_{1,1}$			
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$		
…	…	…	…	
$a_{N-1,0}$	$a_{N-1,1}$	$a_{N-1,2}$	…	$a_{N-1,N-1}$

Matice s nestejně dlouhými řádky

- Prostorově efektivní uložení – opět prvky uložíme do 1-rozměrného pole za sebe (po řádcích):
  - Vytvoříme přístupový vektor – počet položek odpovídá počtu řádků původní matice. Hodnoty udávají součet počtů položek matice ve všech předchozích řádcích (pro tabulku na snímku: [0,2,4,7,16,17]).
  - Mapovací funkce bude mít tvar: $A[i,j] \rightarrow B[PV[i]+j]$

$a_{0,0}$	$a_{0,1}$							
$a_{1,0}$	$a_{1,1}$							
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$						
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$	$a_{3,8}$
$a_{4,0}$								
$a_{5,0}$								

Řídké pole

- Pole, v němž má většina prvků stejnou (dominantní) hodnotu (např. 0).
- Prostorově efektivní uložení:
  - Implementace s použitím vyhledávací tabulky, v níž prvky pole tvoří hodnotu položek tabulky a index(y) tvoří klíč položky tabulky.
  - Ukládáme pouze nedominantní hodnoty.
  - Horší přístupový čas (angl. Access Time) k prvkům pole.
  - Pro vícerozměrné pole lze použít mapovací funkci, která mapuje n-tici indexů do jednoho indexu.

Řídké pole

- InitArr(Arr) -> `InitTable(T)`

- ReadArr(Arr,Ind) ->
```
if Search(T,Ind):
    El ← GetData(T,Ind)
else:
    El ← dominant value
```

- WriteArr(Arr,Ind,El) ->
```
if El = dominant value:
    Delete(T,Ind)
else:
    Insert(T,Ind,El)
```