

Abstraktní datové typy

Karel Richta a kol.

Přednášky byly připraveny s pomocí materiálů, které vyrobili Marko Berezovský, Petr Felkel, Josef Kolář, Michal Píše a Pavel Tvrdík

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

© Karel Richta a kol., 2013

Datové struktury a algoritmy, A7B36DSA
02/2013, Lekce 8

<https://service.felk.cvut.cz/courses/A7B36DSA>



O čem bude řeč?

Obecná charakterizace a způsoby implementace následujících abstraktních datových typů (ADT):

- **Pole (Array)**
- **Zásobník (*Stack*)**
- **Fronta (*Queue*)**
- **Tabulka (*Table*)**
- **Seznam (*List*)**
- **Množina bez opakování (*Set*), s opakováním (*MultiSet, Bag*)**
- **Strom (*Tree*) – později**
- **Graf (*Graph*) – později**

Co je to (datový) typ?

- Každá hodnota zpracovávaná v programu je nějakého **typu**.
- Souhrn použitelných typů je dán užitým **programovacím jazykem**.
- Typ se stanoví při deklaraci proměnné a určuje
 - **množinu (obor) hodnot**, které je možné daným typem vyjádřit
 - **vnitřní reprezentaci** v počítači (velikost paměti, kódování hodnot)
 - **přípustné operace**, které lze nad hodnotami daného typu provádět.

Např. v Pascalu je typ `boolean` :

- množina hodnot je `{true, false}`
 - reprezentace v 1 byte, významný je bit 0 (záleží na implementaci)
 - přípustné logické operace `not`, `and`, `or`
- **Příklady**
 - základní/elementární typy: `char`, `byte`, `int`, `float`, `double`, `uint`, ...
 - reference a ukazatele
 - strukturované typy (`array`, `struct`, `union`, `class`,...)

Abstraktní datový typ (ADT)

- Při tvorbě reálných aplikací lze využít obecného modelu datové struktury vyjádřeného pomocí **abstraktního datového typu**:
 - určíme použité datové komponenty
 - určíme operace a jejich vlastnosti
 - abstrahujeme od způsobu implementace
- **Výhody**
 - ADT je určen tím, co na něm požadujeme/potřebujeme
 - ADT lze implementovat různými způsoby, aniž by to ovlivnilo jeho chování
 - ADT implementujeme pomocí vhodné **datové struktury (DS)**
 - existuje řada často užívaných modelových ADT
- **Příklad**: *bod určený třemi souřadnicemi [x,y,z]* – lze s ním pracovat jako s celkem při programování grafiky


Abstraktní datový typ / / datová struktura

Abstraktní datový typ

- = množina **druhů dat** (hodnot) a **operací**, které jsou přesně specifikovány **nezávisle na konkrétní implementaci**
- reprezentuje **model** složitějšího datového typu
 - je **abstraktní model** - nezávisí na implementaci

Datová struktura

- = konkrétní implementace ADT v daném programovacím jazyce
- zahrnuje reprezentaci druhů dat obsažených v ADT
 - a volbu algoritmů, které implementují operace ADT

Definici ADT lze pojmut  **formálně** (axiomatically) – jako signaturu a axiomy
programátorsky - jako rozhraní (interface) s popisem operací

Příklad: ADT Čítač (rozhraní)

```
public interface Counter {  
    int getValue();  
    void increment();  
    void reset();  
}
```

rozhraní

ADT

+ popis účinku operací na data

```
public class Ctr implements Counter {  
    private int value = 0;  
    public int getValue() { return value; }  
    public void increment() { value++; }  
    public void reset() { value = 0; }  
}
```

Datová
struktura

možná implementace

(pro uživatele je skryta, ten používá jen veřejné metody objektu)

Příklad: ADT Čítač (formální popis)

Signatura

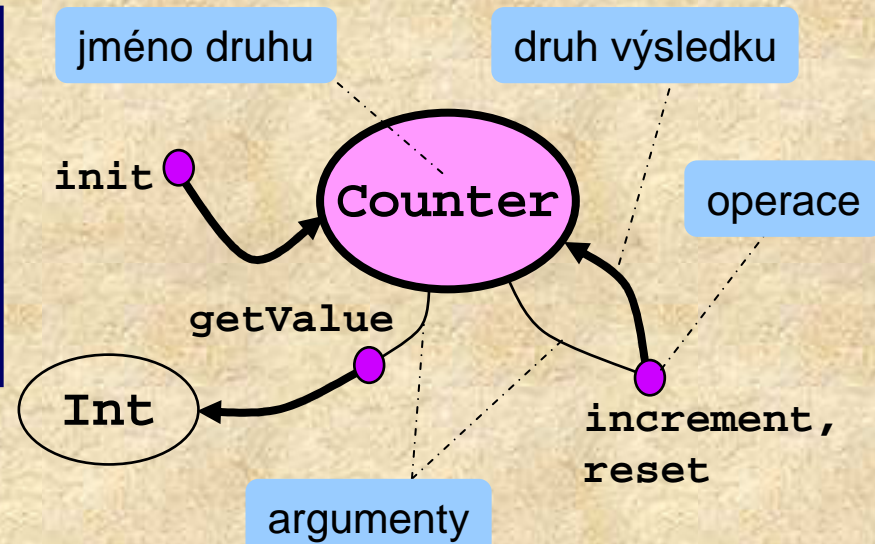
Popisuje **syntaxi**, tj. deklaruje

- **druhy** (jména oborů hodnot) a
- **operace**
 - jméno,
 - druhy (a pořadí) argumentů
 - druh výsledku operace (JEDEN!)

Axiomy

Popisují **vlastnosti operací** (sémantiku) prostřednictvím **ekvivalence výrazů**

Diagram signatury



```
var C: Counter
getValue( init ) = 0
getValue( increment(C) ) = getValue(C) + 1
reset(C) = init
```

Shrnutí

Datový typ

Konkrétní prostředek v daném jazyce

= datová struktura použitelná v konkrétním jazyce

Abstraktní datový typ (ADT)

Matematický model

= množina **druhů dat** (hodnot) a příslušných **operací**, které jsou přesně specifikovány, a to **nezávisle na konkrétní implementaci**.

Datová struktura

Konkrétní implementace ADT v daném jazyce

= realizace (implementace) abstraktního datového typu (pomocí datových typů a konstrukcí daného jazyka)

Základní abstraktní datové typy (ADT)

- Některé abstraktní datové typy se neustále opakují
- Stojí za to je přesně definovat
- Stojí za to je implementovat v knihovnách nebo přímo v jazyce
- Příklady implementace:
 - Balík tříd `java.util`
 - STL (Standard Template Library) šablony v C++

Základní abstraktní datové typy (ADT)

Kontejner (kolekce)

= ADT na organizované skladování objektů podle určitých pravidel
(po implementaci je to např. třída, datová struktura)

Existuje bezprostřední následník?

ANO

Sekvenční (lineární)

- Pole (Array) S
- Zásobník (Stack) D
- Fronta (Queue) D

NE (je nutný klíč)

Asociativní (nelineární)

- Tabulka (Table, Map) S,D
- Množina (Set) D
- Strom (Tree) D

Počet složek: S - statický, D - dynamický \Rightarrow dále **dynamická množina**

Typ složek : stejný (homogenní), různý (nehomogenní) - neprobíráme

Operace nad dynamickou množinou

Modifikující operace (Používají odkaz $x \in S$ na prvek a ne klíč $k \in K$)

- **insert**(x, S) – vloží do množiny S prvek, na který ukazuje x
- **delete**(x, S) – vyjme z množiny S prvek na který ukazuje x

Dotazy (Queries)

- **search**(k, S) – vrací odkaz x na prvek s klíčem k ,
nebo **nil**, pokud k v S není

Pro úplně uspořádané klíče (pro lib. $a, b \in K$ platí právě jedno: $a < b$, $a = b$, nebo $a > b$)

- **min**(S) – vrací prvek s nejmenším klíčem
- **max**(S) – vrací prvek s největším klíčem
- **succ**(x, S) – pro prvek x vrací prvek s nejbližše vyšším klíčem,
nebo **nil** pro největší prvek
- **pred**(x, S) – pro prvek x vrací prvek s nejbližše nižším klíčem,
nebo **nil** pro nejmenší prvek

Slovník (Dictionary)

Další operace nad dynamickou množinou

Konstruktor

- `init(S)` – vytvoří a inicializuje kontejner

Modifikující operace

- `clear(S)` – vymaže všechny prvky v množině S

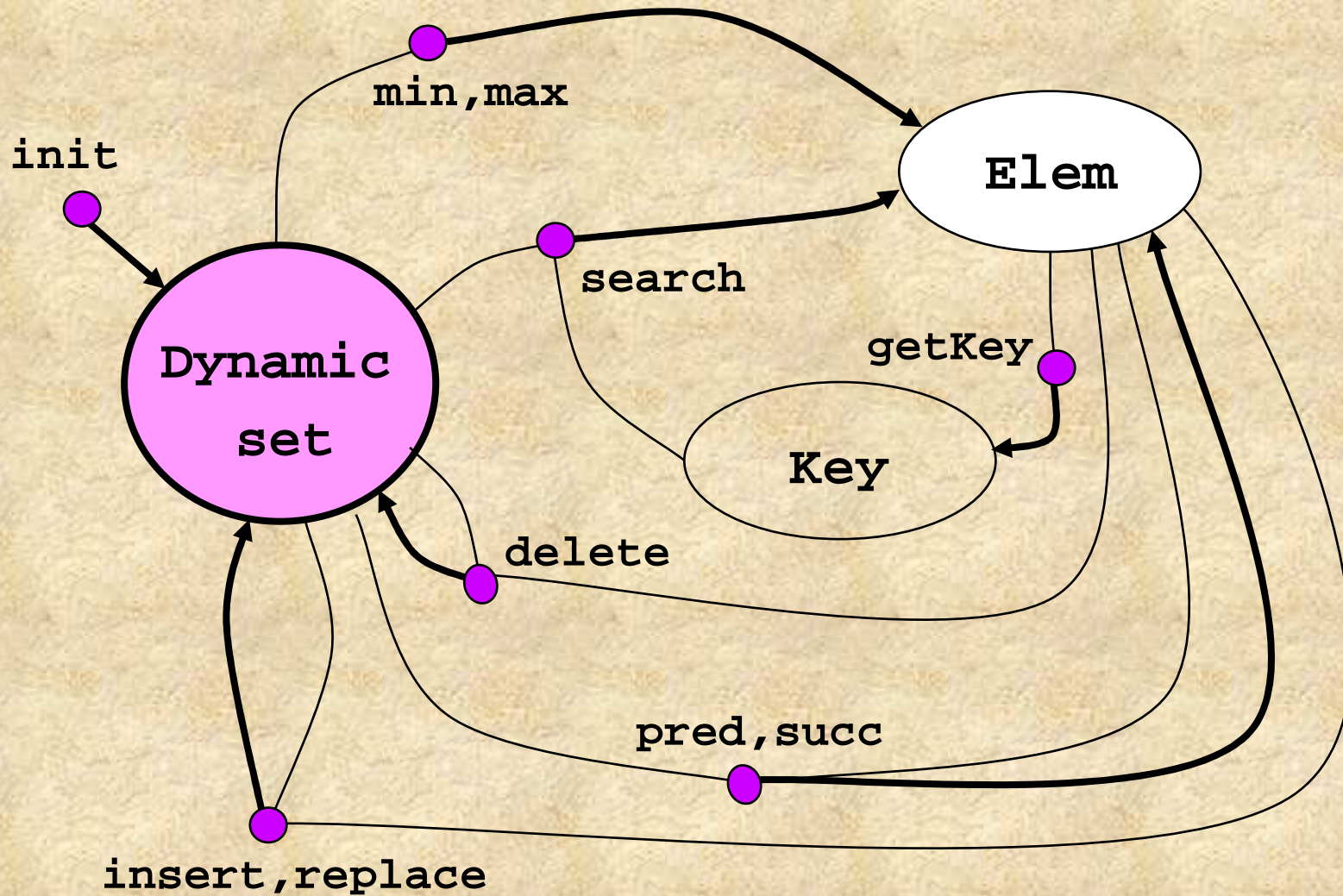
Dotazy (Queries)

- `size(S)` – vrátí počet prvků v S

Predikáty (vracejí `true` / `false`)

- `empty(S)` – vrátí logickou hodnotu, zda je množina S prázdná
- `full(S)` – vrátí logickou hodnotu, zda je množina S plná
(nutné při konkrétní implementaci)

Signatura dynamické množiny



Abstraktní datové typy

✓ Pole (*Array*)

- Zásobník (*Stack*)
- Fronta (*Queue*)
- Tabulka (*Table*)

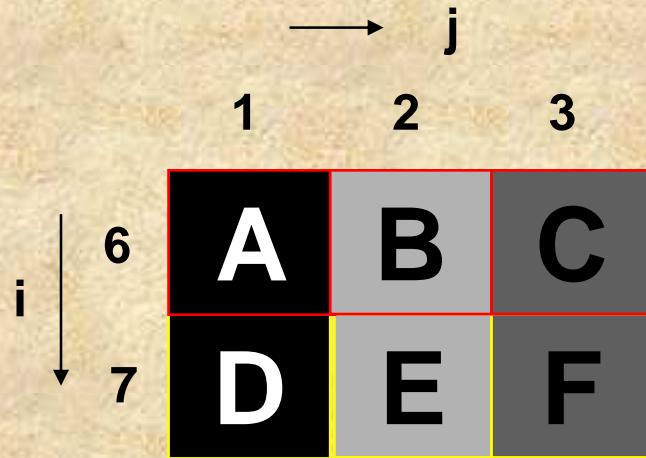
- Množina bez opakování (*Set*)
- Množina s opakováním (*MultiSet*)
- Seznam (*List*)

Pole (Array)

	1	2	3
6	A	B	C
7	D	E	F

- **VELMI frekventovaný** ADT
- \Rightarrow patří mezi typy poskytované běžnými prog. jazyky
- paměť počítače je také pole – jednorozměrné (1D)
- \Rightarrow všechny datové struktury se vlastně mapují do 1D pole
- ukážeme hlavně pro 2D, 3D, ..., nD pole

Pole (Array)



Pole o rozměrech 2 x 3 prvky

- řádkový index $i \in \{6, 7\}$
- sloupcový index $j \in \{1, 2, 3\}$

$a[6, 3] \rightarrow C$ buňka pole na řádku 6, a sloupci 3

Pole (Array)

	1	2	3
6	A	B	C
7	D	E	F

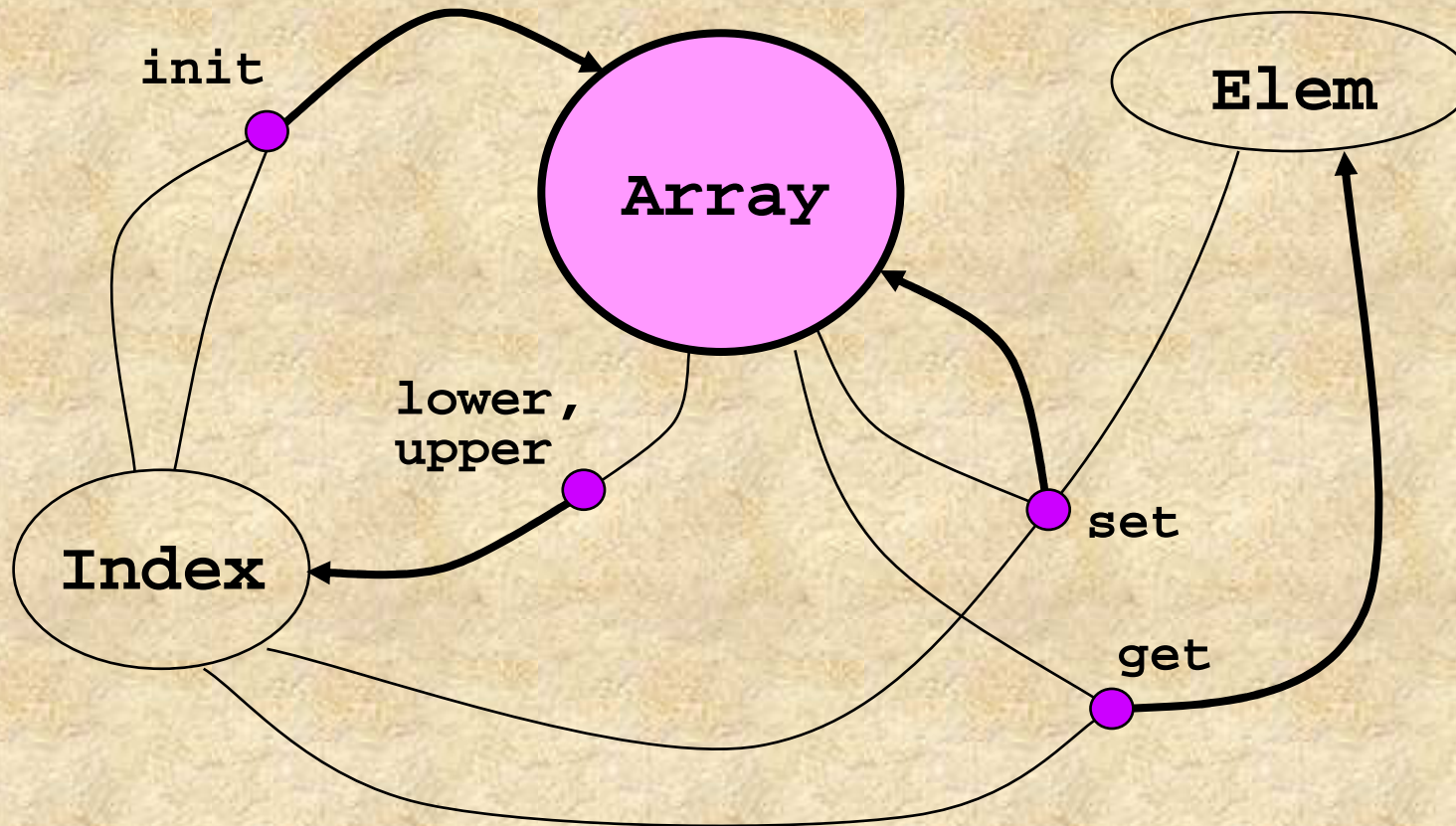
Vlastnosti

- prvky jsou **stejného** typu – *homogenní ADT*
- všechny prvky jsou **současně** v paměti
- \Rightarrow umožňuje rychlý **náhodný přístup** (*random-access*), pozici určují **indexy**
- **známý počet** prvků – *statický ADT*
- indexy jsou uspořádány – *lineární ADT*
- je dán počet dimenzí (n) a meze indexů
- přístup k prvkům – pomocí tzv. **mapovací funkce**

`a[6,3] -> C`

Signatura 1D pole

	1	2	3
6	A	B	C
7	D	E	F



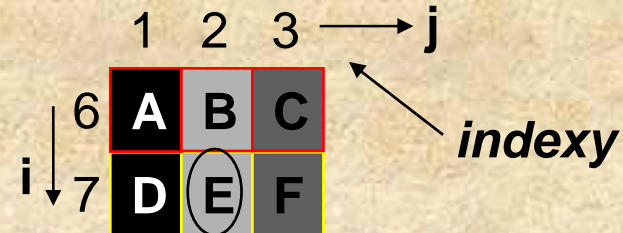
Implementace pole – mapovací funkce

	1	2	3
6	A	B	C
7	D	E	F

Logická struktura

array $a[6..7, 1..3]$

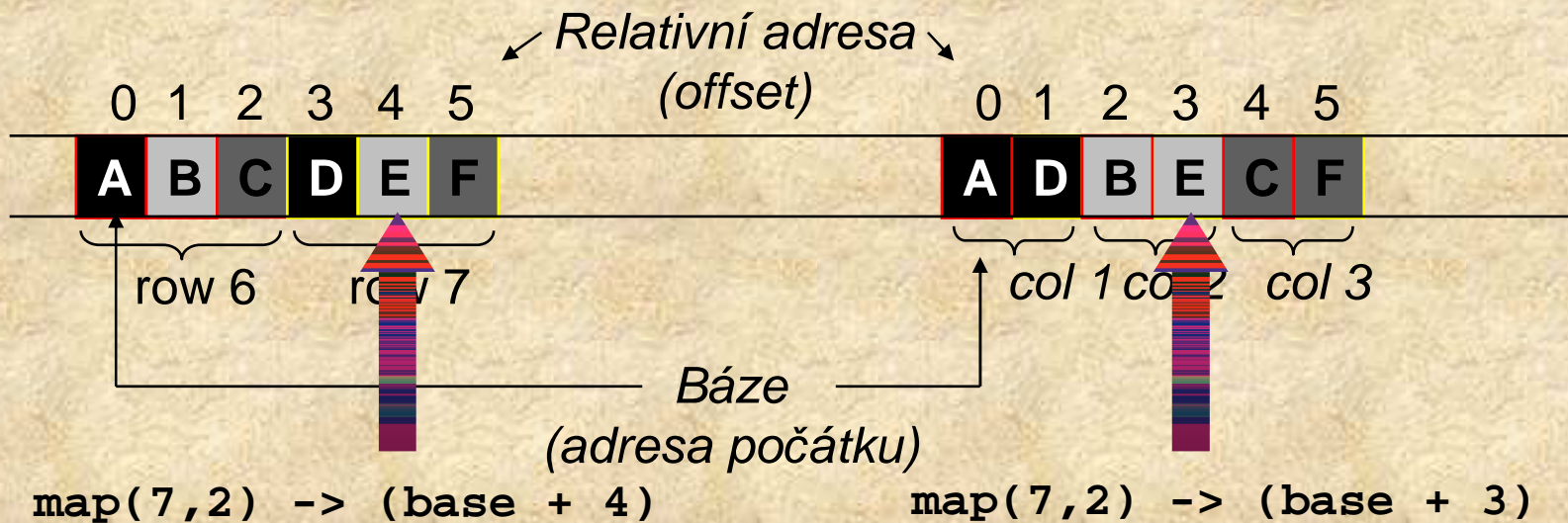
Příklad: $a[7,2] \rightarrow E$



Fyzické umístění v paměti

Uložení po řádcích

Uložení po sloupcích



Implementace pole – mapovací funkce

	1	2	3
6	A	B	C
7	D	E	F

0	1	2	3	4	5
A	B	C	D	E	F

Uložení po řádcích pro 2D

Adresa počátku pole (Báze) Řádkový offset (# řádek k přeskočení) Délka řádky = počet sloupců

$$\text{map}(i,j) = a(i_{\min}, j_{\min}) + (i-i_{\min})n_j + (j-j_{\min})$$

$$\text{map}(i,j) = a(0,0) + (i*n_j + j) \text{ pro indexy od } 0$$

Sloupcový offset

$$n_j = j_{\max} - j_{\min} + 1$$

Relativní adresa prvku v 1D poli

Pro naše pole: $\text{map}(i, j) = a(6,1) + (i-6)*3 + (j-1)$
 $\text{map}(7,2) = a(6,1) + (7-6)*3 + (2-1)$
 $= a(6,1) + 4$

	1	2	3
6	A	B	C
7	D	E	F

0	1	2	3	4	5
A	B	C	D	E	F

Implementace pole – mapovací funkce

	1	2	3
6	A	B	C
7	D	E	F

0	1	2	3	4	5
A	B	C	D	E	F

Uložení po řádcích pro 3D

adresa báze

vrstva

řádek

sloupec

$$\text{map}(i,j,k) = a(i_{\min}, j_{\min}, k_{\min}) + (i-i_{\min}) n_j n_k + (j-j_{\min}) n_k + (k-k_{\min})$$

$$\text{map}(i,j,k) = a(0,0,0) + (i * n_j + j) * n_k + k \text{ pro indexy od } 0$$

relativní adresa prvku
(element offset)

Implementace pole – mapovací funkce

	1	2	3
6	A	B	C
7	D	E	F

0	1	2	3	4	5
A	B	C	D	E	F

Uložení po sloupcích pro 2D

adresa počátku pole (Báze) sloupcový offset (# řádek k přeskočení) délka sloupce = počet řádků

$$\text{map}(i,j) = a(i_{\min}, j_{\min}) + (j-j_{\min})n_i + (i-i_{\min})$$

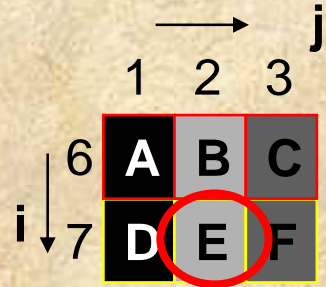
$$\text{map}(i,j) = a(0,0) + (j*n_i + i) \text{ pro indexy od } 0$$

řádkový offset

$$n_i = i_{\max} - i_{\min} + 1$$

Relativní adresa prvku v 1D poli

Pro naše pole: $\text{map}(i, j) = a(6,1) + (j-1)*2 + (i-6)$
 $\text{map}(7,2) = a(6,1) + (2-1)*2 + (7-6)$
 $= a(6,1) + 3$



0	1	2	3	4	5
A	D	E	C	F	

Implementace pole – mapovací funkce

	1	2	3
6	A	B	C
7	D	E	F

0	1	2	3	4	5
A	B	C	D	E	F

Uložení po sloupcích pro 3D

adresa báze

$$\text{map}(i,j,k) = a(i_{\min}, j_{\min}, k_{\min}) + (k-k_{\min}) n_j n_i + (j-j_{\min}) n_i + (i-i_{\min})$$
$$\text{map}(i,j,k) = a(0,0,0) + (k*n_j + j) * n_i + i \text{ pro indexy od 0}$$

relativní adresa prvku
(element offset)

Pole - shrnutí

	1	2	3
6	A	B	C
7	D	E	F

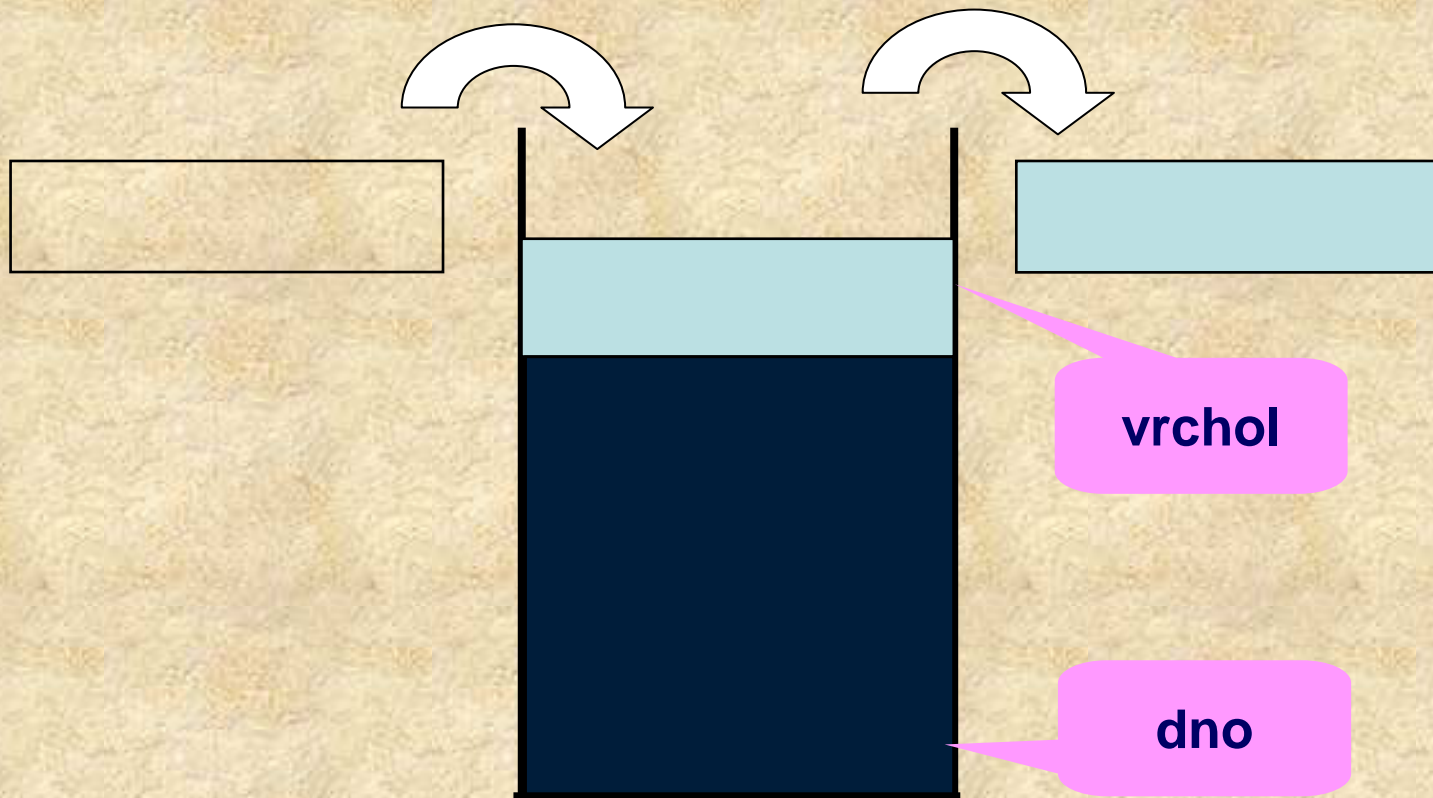
- nejpoužívanější ADT v počítačích (i paměť je pole)
- pevný počet prvků (při vytváření dán počet dimenzí n a meze indexů)
- všechny prvky *současně* v paměti
- rychlý *náhodný přístup* (*random-access*) → *INDEX*
pomocí mapovací funkce $map(indexy) \rightarrow adresa$
- všechny prvky *stejného* typu – *homogenní*
- je *známý* počet prvků – *statické*
- indexy uspořádány – *lineární*

Abstraktní datové typy

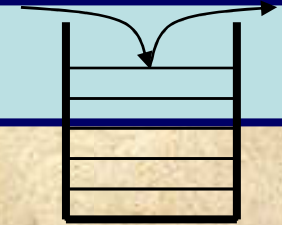
- Pole (*Array*)
- ✓ **Zásobník (*Stack*)**
- Fronta (*Queue*)
- Tabulka (*Table*)

- Množina bez opakování (*Set*)
- Množina s opakováním (*MultiSet*)
- Seznam (*List*)

Zásobník (Stack)



Zásobník (Stack)



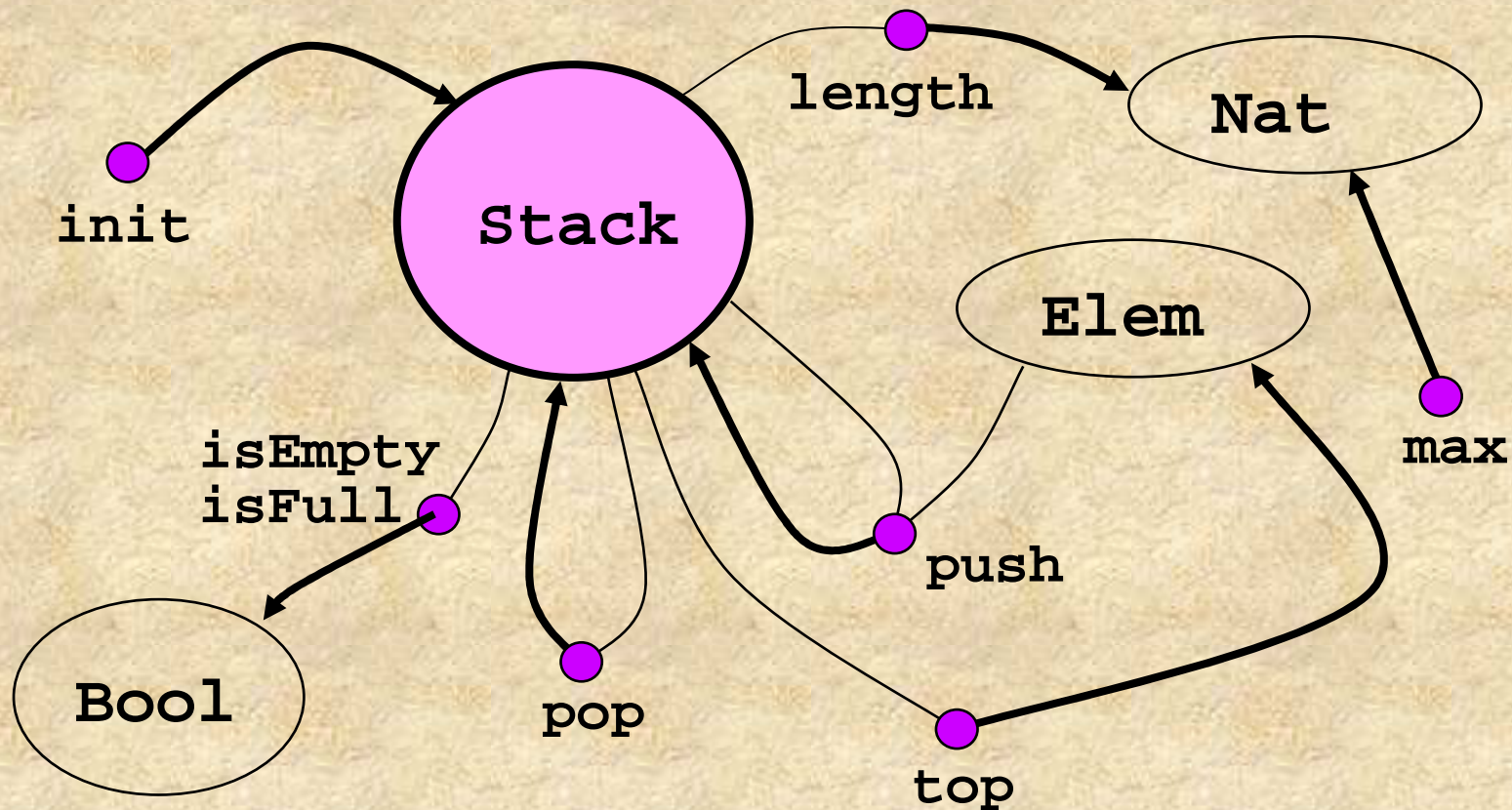
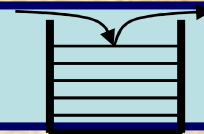
Použití

- odložení informace, výběr v opačném pořadí (návrat z procedury, uzlové body cesty, náboje v pistoli, průchod stromem do hloubky,...)

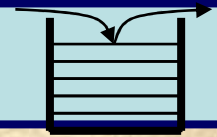
Vlastnosti

- LIFO = *Last-in, First-out* („poslední tam, první ven“)
- přístup pouze k prvku *na vrcholu (top)*
- vkládání pouze na vrchol (*top*)
- *homogenní, lineární, dynamický*

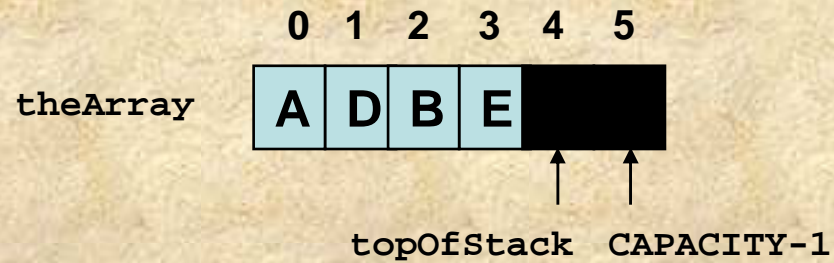
Signatura zásobníku



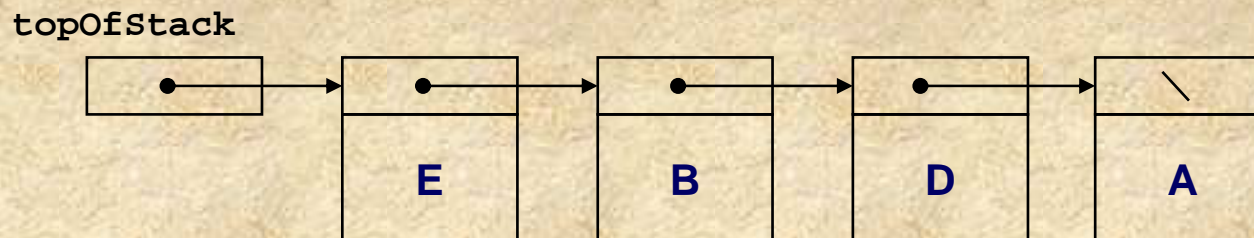
Implementace zásobníku



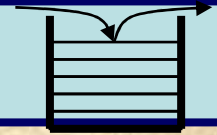
použitím pole



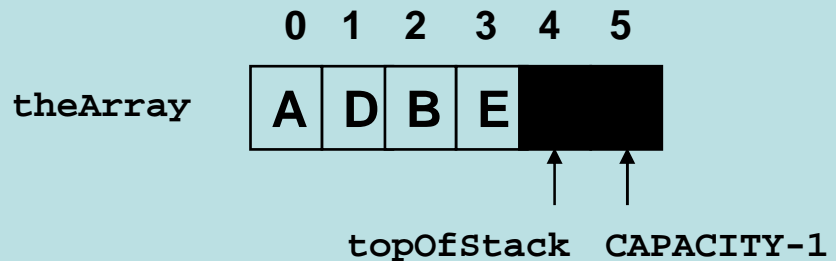
použitím dynamické paměti



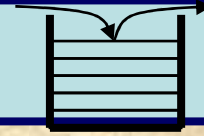
Implementace zásobníku v poli



```
// *****PUBLIC OPERATIONS*****  
// void push( x )      --> Insert x  
// void pop( )        --> Remove most recently inserted item  
// Object top( )      --> Return most recently inserted item  
// boolean isEmpty( ) --> Return true if empty; else false  
// boolean isFull( )  --> Return true if full; else false  
// void init( )       --> Remove all items  
// int max( )         --> Return stack capacity  
// int length( )      --> Return actual # of elements in stack  
public interface Stack {  
    void    push( Elemt x );  
    void    pop( );  
    Object  top( );  
    boolean isEmpty( );  
    boolean isFull( );  
    void    init( );  
    int     max( );  
    int     length( );  
}
```

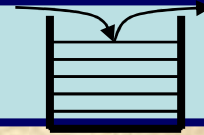


Implementace zásobníku v poli



```
public class ArrayStack implements Stack {  
  
    public ArrayStack( ) {  
        theArray = new Object[ CAPACITY ];  
        topOfStack = -1  
    }  
  
    public boolean isEmpty( ) {  
        return topOfStack == -1;  
    }  
  
    public void init( ) {  
        topOfStack = -1;  
    }  
  
    public Object top( ) {  
        if( isEmpty( ) )  
            throw new StackException( "ArrayStack top" );  
        return theArray[ topOfStack ];  
    }  
  
    public void pop( ) {  
        if( isEmpty( ) )  
            throw new StackException( "ArrayStack pop" );  
        topOfStack--;  
    }  
  
}
```

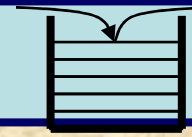
Implementace zásobníku v poli



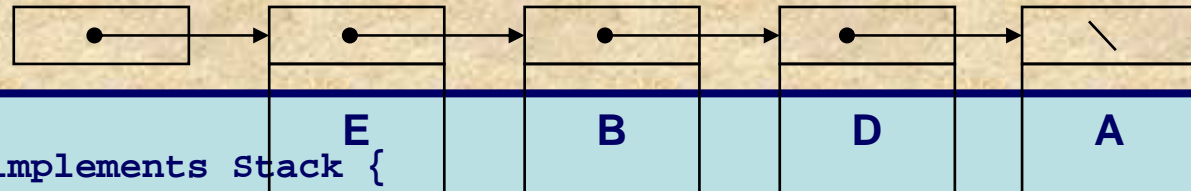
```
public void push( Object x ) {
    if( topOfStack == CAPACITY - 1 )
        throw new StackException( "ArrayStack push" );
    theArray[ ++topOfStack ] = x;
}
boolean isFull( ) { ... // doplnte sami jako cviceni
}
int max( ) { ... // doplnte sami jako cviceni
}
int length( ) { ... // doplnte sami jako cviceni
}
private Object [ ] theArray;
private int topOfStack;
private static final int CAPACITY = 10;
}
public class StackException extends RuntimeException {
    public StackException( String message ) {
        super( message );
    }
}
}
```

Jaká je výpočetní složitost jednotlivých operací ?

Zásobník v dynamické paměti



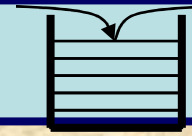
topOfStack



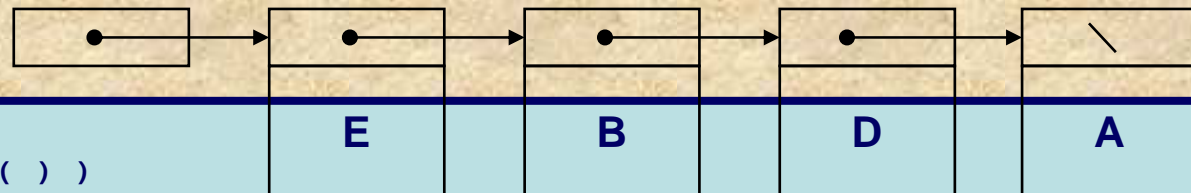
```
public class ListStack implements Stack {  
    public ListStack( ) {  
        topOfStack = null;  
    }  
    public boolean isEmpty( ) {  
        return topOfStack == null;  
    }  
    public void init( ) {  
        topOfStack = null;  
    }  
    public void push( Object x ) {  
        topOfStack = new ListNode( x, topOfStack );  
    }  
    public void pop( ) {  
        if( isEmpty( ) )  
            throw new StackException( "ListStack pop" );  
        topOfStack = topOfStack.next;  
    }  
}
```

**Jaká je
výpočetní
složitost
jednotlivých
operací ?**

Zásobník v dynamické paměti



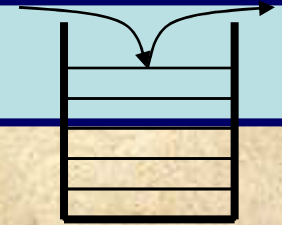
topOfStack



```
public Object top( ) {  
    if( isEmpty( ) )  
        throw new StackException( "ListStack top" );  
    return topOfStack.element;  
}  
boolean isFull( ) { ...  
}  
int max( ) { ...  
}  
int length( ) { ...  
}  
private ListNode topOfStack;  
}
```

```
class ListNode {  
    public ListNode( Object theElement, ListNode n ) {  
        element = theElement; next = n;  
    }  
    public Object element;  
    public ListNode next;  
}
```

Zásobník – Shrnutí



Vlastnosti

- LIFO = *Last-in, First-out* („poslední tam, první ven“)
- přístup pouze k prvku *na vrcholu (top)*
- vkládání pouze na vrchol (*top*)
- *homogenní, lineární, dynamický*

Abstraktní datové typy

- Pole (*Array*)
- Zásobník (*Stack*)
- ✓ **Fronta (*Queue*)**
- Tabulka (*Table*)

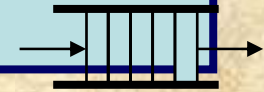
- Množina bez opakování (*Set*)
- Množina s opakováním (*MultiSet*)
- Seznam (*List*)

Fronta (*Queue*)



Čelo

Fronta (*Queue*)



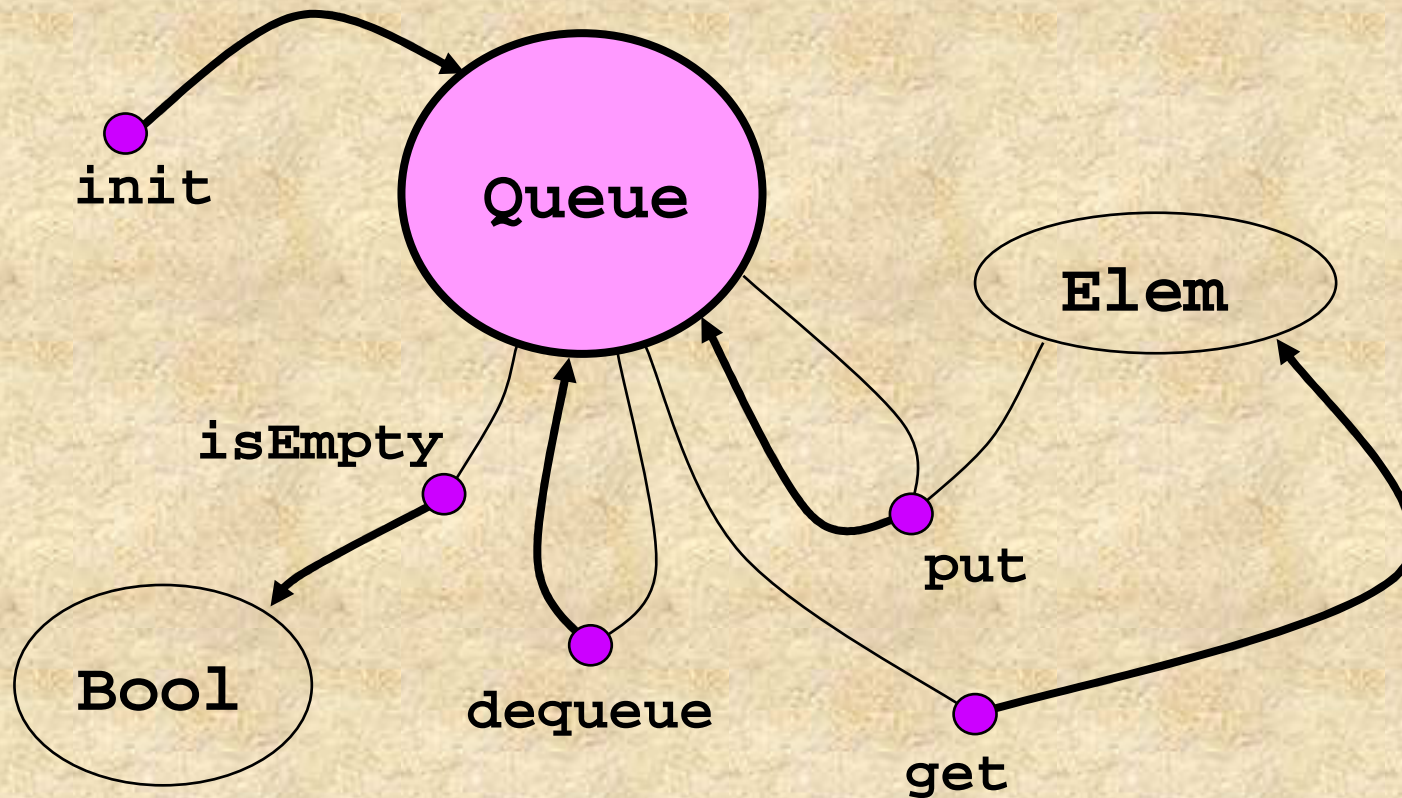
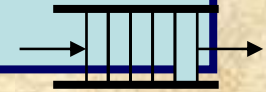
Použití

- Odložení informace, výběr ve stejném pořadí, jako se vkládalo
- Průchod stromem či grafem do šířky – algoritmus vlny,...
- Obsluha sdílených prostředků (fronta na tiskárnu, fronta na pivo,...)

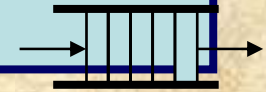
Vlastnosti

- FIFO = *First-in, First-out* („Kdo dřív přijde, ten dřív mele“)
- přístup pouze k prvku *na* začátku (čelo, head)
- vkládání pouze na konec fronty (konec, tail)
- *homogenní, lineární, dynamická*

Signatura fronty

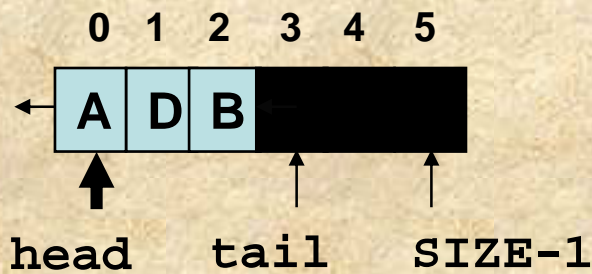


Implementace fronty

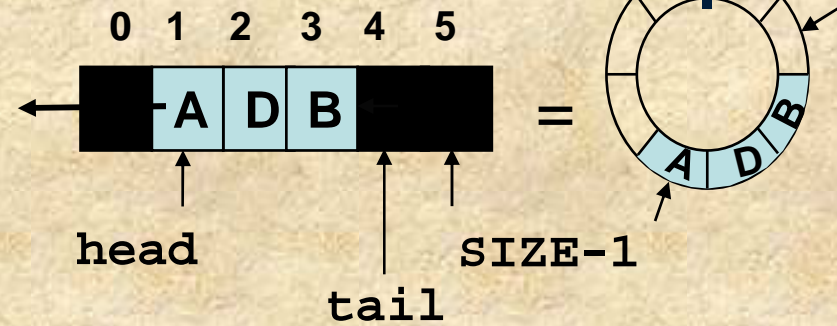


pomocí pole

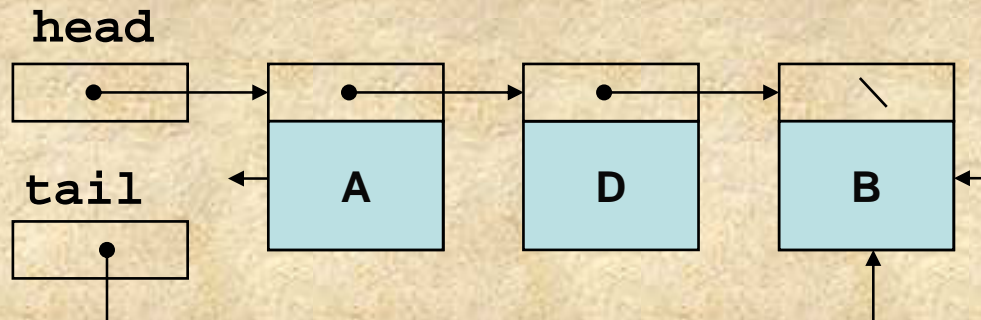
lineární (naivní)



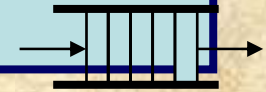
kruhová



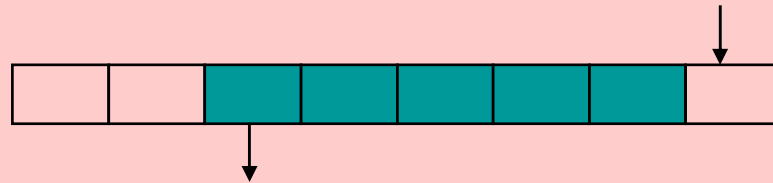
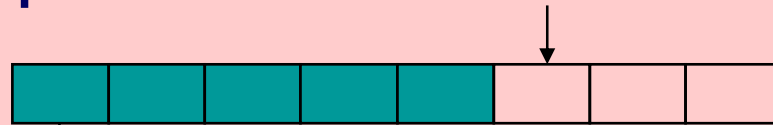
pomocí dynamické paměti



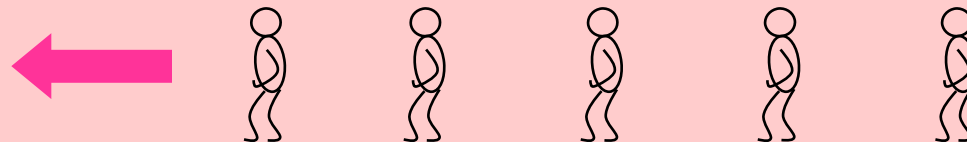
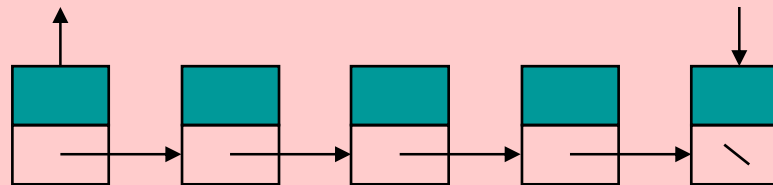
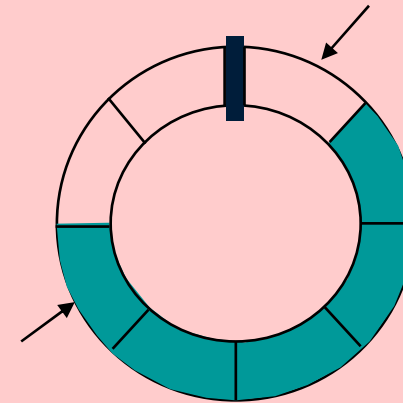
Typické implementace fronty



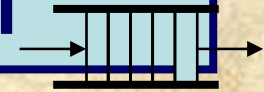
Různé implementace



=

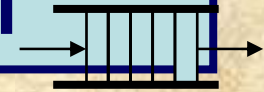


Implementace fronty v kruhovém poli



```
// *****PUBLIC OPERATIONS*****
// void put( x )          --> Insert x
// Object get( )         --> Return least recently inserted item
// void dequeue( )       --> Remove least recent item
// boolean isEmpty( )    --> Return true if empty; else false
// void init( )          --> Remove all items
public class ArrayQueue implements Queue {
    public ArrayQueue( ) {
        theArray = new Object[ CAPACITY ];
        init( );
    }
    public boolean isEmpty( ) {
        return currentSize == 0;
    }
    public void init( ) {
        currentSize = 0;
        head = 0;
        tail = -1;
    }
}
```

Implementace fronty v kruhovém poli



```
public void dequeue( ) {
    if( isEmpty( ) )
        throw new QueueException( "ArrayQueue delFront" );
    currentSize--;
    head = increment( head );
}
```

```
public Object get( ) {
    if( isEmpty( ) )
        throw new QueueException( "ArrayQueue front" );
    return theArray[ head ];
}
```

```
public void put( Object x ) {
    if( currentSize == CAPACITY )
        throw new QueueException( "ArrayQueue insLast" );
    back = increment( tail );
    theArray[ tail ] = x;
    currentSize++;
}
```

Implementace fronty v kruhovém poli



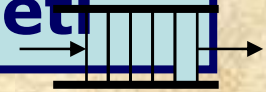
```
private int increment( int x ) {
    if( ++x == CAPACITY )
        x = 0;
    return x;
}

private Object [ ] theArray;
private int      currentSize;
private int      head;
private int      tail;

private static final int CAPACITY = 10;
}

public class QueueException extends RuntimeException {
    public QueueException( String message ) {
        super( message );
    }
}
```

Implementace fronty v dynamické paměti



```
public class ListQueue implements Queue {
    public ListQueue( ) {
        head = tail = null;
    }
    public boolean isEmpty( ) {
        return head == null;
    }
    public void put( Object x ) {
        if( isEmpty( ) )           // Make queue of one element
            tail = head = new ListNode( x );
        else                       // Regular case
            tail = tail.next = new ListNode( x );
    }
    public Object dequeue( ) {
        if( isEmpty( ) )
            throw new QueueException( "ListQueue dequeue" );
        Object returnValue = head.element;
        head = head.next;
        return returnValue;
    }
}
```

Implementace fronty v dynamické paměti



```
public Object get( ) {
    if( isEmpty( ) )
        throw new UnderflowException( "ListQueue get" );
    return head.element;
}

public void init( ) {
    head = null;
    tail = null;
}

private ListNode head;
private ListNode tail;
}

class ListNode {           // viz dynamicka implementace zasobniku
    ...
}
```

Fronta – Shrnutí



- použití na úlohy hromadné obsluhy, procesy v počítači, ...
- FIFO = *First-in, First-out* („Kdo dřív přijde, ten dřív mele“)
- přístup pouze k prvnímu prvku (*head*)
- vkládání pouze na konec (*tail*)
- homogenní, lineární, dynamická

Axiomatická sémantika

Dosud jsme nepoužili formální (matematický, axiomatický) způsob popisu operací v ADT.

Signaturu lze místo obrázku vyjádřit přesněji i jinak (existují definiční jazyky Claude, Maude a další):

Příklad ADT Bool (logická hodnota)

Druhy:

`Bool`

Operace:

```
true, false: Bool      (konstanty, nulární operace)
not(_): Bool -> Bool  (unární operace)
and(_, _): Bool, Bool -> Bool (binární op.)
or(_, _) : Bool, Bool -> Bool
```


ADT Logická hodnota

Pomocí operaci ADT můžeme vytvářet smysluplné **výrazy**:

- stav instance ADT popsán výrazem, který ho zkonstruuje
- výraz
 - se skládá z názvů operací a proměnných
 - lze zjednodušit, pokud najdu odpovídající axiom ("pravidlo")
 - porovnává se textově (pattern matching)
 - stejný stav ADT lze popsat více výrazy
- axiom má podobu rovnosti výrazů

Příklad:

axiom `not(true) = false`

znamená "místo `not(true)` lze psát `false`"

Sémantika ADT Logická hodnota

`not(true) = false` (1)

`not(false) = true` (2)

negace

`and(x, true) = x` (3)

`and(x, false) = false` (4)

`and(x, y) = and(y, x)` (5)

logický součin AND

`or(x, true) = true` (6)

`or(x, false) = x` (7)

`or(x, y) = or(y, x)` (8)

logický součet OR

ADT Logická hodnota

Ukázky úpravy výrazů - cílem je co nejjednodušší výraz

`not(not(true)) = not(false) =` neboť `not(true)=false (1)`
 `= true` neboť `not(false)=true (2)`

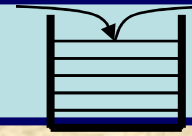
`and(or(x,true), or(y, false)) =` podle (6)
 `= and(true, or(y, false)) =` podle (7)
 `= and(true, y) =` podle (5)
 `= and(y, true) = y` podle (3)

`and(not(x),not(y))`

neumíme pomocí daných axiomů upravit, museli bychom doplnit axiom:

`and(not(x), not(y)) = not(or(x,y))`

ADT Stack (Zásobník)



Jak by to vypadalo pro zásobník

Operations:

`init: -> Stack`

`isEmpty(_): Stack -> Bool`

`push(_, _): Elem, Stack -> Stack`

`top(_): Stack -> Elem`

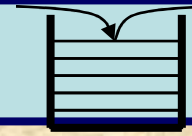
`pop(_): Stack -> Stack`

`length(_): Stack -> Nat`

`max: -> Nat`

`isFull(_): Stack -> Bool ... omezení počtu prvků`

ADT Stack (Zásobník)



Axiomy zásobníku:

```
isEmpty( init ) = true
```

```
isEmpty( push( e, s ) ) = false
```

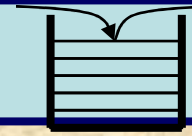
```
top( init ) = error_elem
```

```
top( push( e, s ) ) = e
```

```
pop( init ) = init
```

```
pop( push( e, s ) ) = s
```

ADT Stack (Zásobník)



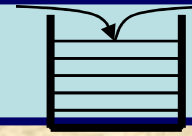
1. `isEmpty(init()) = true`
2. `isEmpty(push(e, s)) = false`

```
isEmpty( push("X", push("Y", pop( push( "A", init() ) ))) ) =  
= false
```

```
isEmpty( pop( push( "A", init() ) ) ) = ???
```

... potřebujeme další axiomy

ADT Stack (Zásobník)



3. `top(init()) = error_elem()`

4. `top(push(e, s)) = e`

```
top( push("X", push("Y", pop( push( "A", init() ) ) ) ) ) =  
= "X"
```

```
top( init() ) = error_elem
```

```
top( pop( push( "A", init() ) ) ) = ???
```

... opět potřebujeme další axiomy

5. `pop(init) = init()`

6. `pop(push(e, s)) = s`

```
pop( push("A", init() ) ) = init()
```

```
pop( push("X", push("Y", pop( push( "A", init() ) ) ) ) )
```

```
= push("Y", pop( push( "A", init() ) ) )
```

```
= push("Y", init() )
```

Abstraktní datové typy

- Pole (*Array*)
- Zásobník (*Stack*)
- Fronta (*Queue*)
- ✓ **Tabulka (*Table*)** příště
- Množina bez opakování (*Set*)
- Množina s opakováním (*MultiSet*)
- Seznam (*List*)

Prameny

- **Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press, 1990**
- Jan Honzík: Programovací techniky, skripta, VUT Brno, 19xx
- Karel Richta: Datové struktury, skripta pro postgraduální studium, ČVUT Praha, 1990
- Bohuslav Hudec: Programovací techniky, skripta, ČVUT Praha, 1993
- Miroslav Beneš: Abstraktní datové typy, Katedra informatiky FEI VŠB-TU Ostrava, <http://www.cs.vsb.cz/benes/vyuka/upr/texty/adt/index.html>

References

- Steven Skiena: The Algorithm Design Manual, Springer-Verlag New York, 1998
<http://www.cs.sunysb.edu/~algorith>
- Code examples: M.A.Weiss: Data Structures and Problem Solving using JAVA, Addison Wesley, 2001, code web page:
<http://www.cs.fiu.edu/~weiss/dsj2/code/code.html>
- Paul E. Black, "abstract data type", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., [U.S. National Institute of Standards and Technology](http://www.nist.gov/dads/HTML/abstractDataType.html). 10 February 2005. (accessed 10.2006) Available from:
<http://www.nist.gov/dads/HTML/abstractDataType.html>
- "Abstract data type." [Wikipedia, The Free Encyclopedia](http://en.wikipedia.org/w/index.php?title=Abstract_data_type&oldid=78362071). 28 Sep 2006, 19:52 UTC. Wikimedia Foundation, Inc. 25 Oct 2006
http://en.wikipedia.org/w/index.php?title=Abstract_data_type&oldid=78362071