Datové struktury a algoritmy

Část 4

Abstraktní datové typy

Maude inside

Petr Felkel

27.10. 2007

Abstraktní datové typy

Co nás čeká?

Vyjasnění pojmů

- Datový typ
- Abstraktní datový typ (ADT)
- Datová struktura

- Syntaxe
- Sémantika

Základní ADT

Základní abstraktní datové typy (ADT)

Fronta (*Queue*)

Zásobník (*Stack*)

Pole (*Array*)

Tabulka (*Table*)

Seznam *(List)*

Strom *(Tree)*

Množina (Set)

Graf (Graph)

…

Datové typy v programovacích jazycích

Datový typ (nebo jen typ)

- Poskytován konkrétním *programovacím jazykem*
- Pojmenování pro *množinu hodnot* a sadu *operací*
- Používá ke (statické či dynamické) kontrole programů
  aby se nesčítaly hrušky s jablky, jako 7+“ahoj“

Příklady:

- Primitivní typy - char, byte, int, float, double, …, pole
- Složené typy - záznam (struct), třída(class),
- Reference (struct tree *left, *right), a typ funkce
- (Abstraktní datové typy - zásobník, pole, strom, …)

Datové typy v programovacích jazycích

Příklad z C/C++

```
int i=7, j=5, k=0; float m;
```

“Ví se“, že (dáno definicí jazyka):

```
m = "ahoj";     // nepřeloží se
                   - statická kontrola
m = i / j;      // m bude 1 a ne 1.4
                   - celočíselné dělení
m = i / k;      // chyba, dělení nulou
...
```

Abstraktní datové typy

Pro složitější projekty nestačí základní datové typy

- nezapouzdřují (lze porušit jejich integritu)
- jsou málo abstraktní (mozek zvládá jen určité penzum složitosti a proto má rád abstrakci)

Proto komplexnější *abstraktní datové typy (ADT).*

Příklad: bod jako 3 souřadnice – lze s ním pracovat jako s celkem při programování grafiky

Abstraktní datové typy

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++

Abstraktní datový typ

Abstraktní datový typ (ADT)

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

Definovat lze

- Matematicky – signatura a axiomy
- Jako rozhraní (interface) s popisem operací
  - Interface poskytuje
    - konstruktor, který vrací abstraktní odkaz a
    - operace, které akceptují odkaz jako argument a které mají přesně definovaný účinek na data

Příklad 1: Čítač

Čítač jako rozhraní

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

+ popis vlastností operací

Rozhraní

ADT

```
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

Viz dále

Možná implementace

Pro uživatele je implementace skryta, používá jen veřejné metody objektu

Abstraktní datový typ

SYNTAXE

… signatura

= deklarace druhů

- Jména oborů hodnot

\+ deklarace operací

- Jména operací
- Druhy (a pořadí) argumentů operací
- Druh výsledku (JEDEN!)

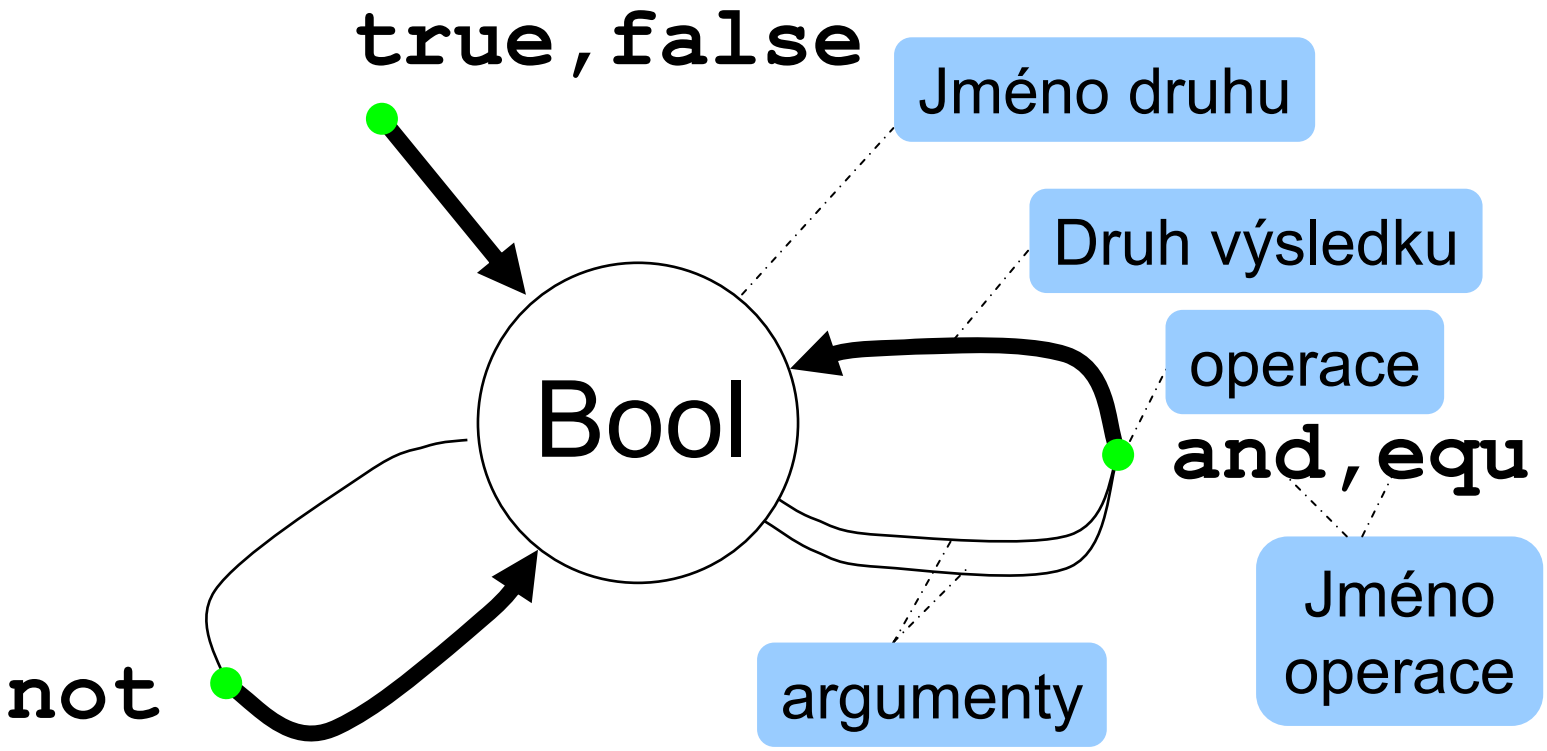
SÉMANTIKA

… axiomy

= popis vlastností operací!

Příklad 2: Logická hodnota

Signatura pomocí diagramu



Příklad 2: Logická hodnota

Signatura symbolicky (jazyk Clear, Maude)

Výhoda: Je jasné pořadí argumentů

Druhy:

Bool

Operace:

`true, false: Bool` (konstanty, nulární operace)

`not(_): Bool -> Bool` (unární operace)

`and(_,_): Bool,Bool -> Bool` (binární op.)

`equ(_,_) : Bool,Bool -> Bool`

Jméno operace | (_,...) | : | Druhy argumentů | -> | Druh výsledku

Příklad 2: Logická hodnota

Notace a priorita operací

Prefixová notace

```
and(_,_): Bool,Bool -> Bool.
```
… Clear, Maude

```
and : Bool,Bool -> Bool.
```
… Maude

pořadí operací dáno uzávorkováním

Infixová notace

and: Bool,Bool -> Bool

priorita: not,and ... Priorita operací se musí explicitně

Příklad 2: Logická hodnota

Sémantika pomocí axiomů

Stav instance ADT popsán *výrazem*, který ho zkonstruuje

Výraz

- se skládá z názvů operací a proměnných
- lze zjednodušit, pokud najdu odpovídající axiom
- porovnává se textově (pattern matching)
- stejný stav lze popsat více výrazy

Axiom je rovnost výrazů

př. not(true) = false znamená

místo not(true) *piš* false

Příklad 2: Logická hodnota

Sémantika pomocí axiomů

```
var x,y: Bool
```

```
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

Viz dále

```
or(x, true) = true      (6)
or(x,false) = x         (7)
or(x,y) = or(y,x)       (8)
```

logický součet OR

v Maude [comm]

Příklad 2: Logická hodnota

Test na rovnost - verze 1 – bez proměnných – na výrazy nepoužitelná

```
equ(true, true) = true        (9)
equ(true,false) = false      (10)
equ(false, true) = false     (11)
equ(false,false) = true      (12)
```

Test na rovnost - verze 2 – je lepší, použitelná na výrazy jejichž hodnotu ještě neznáme

```
equ(x, true) = x              (13)
equ(x, false) = not(x)        (14)
equ(x, y) = equ(y, x)         (15)
```

v Maude [comm]

Příklad 2: Logická hodnota

Příklad úpravy výrazů

Cílem je co nejjednodušší výraz

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

and(or(x,true), or(y, false)) $\overset{(6)}{=}$ and(true, or(y, false)) $\overset{(7)}{=}$ and(true, y) $\overset{(5)}{=}$

= and(y, true) $\overset{(3)}{=}$ y

and(not(x),not(y)) = … neumím s danými axiomy přímo upravit,

leda bych doplnil axiom: and(not(x), not(y)) = not(or(x,y))

Příklad 2: Logická hodnota v Maude

```
fmod BOOL is                                          SIGNATURA
  sort Bool .

  op true : -> Bool  [ctor].        … konstruktor (konec redukce)
  op false : -> Bool [ctor].

  op not : Bool -> Bool .
  op and : Bool Bool -> Bool  [comm] .
  op or  : Bool Bool -> Bool  [comm] .
  op equ : Bool Bool -> Bool  [comm] .
```

Příklad 2: Logická hodnota v Maude

```
vars X Y Z : Bool .
 AXIOMY


eq not(true) = false .
eq not(false) = true .


eq and(X, false) = false .
eq and(X, true) = X .
eq and(X, X) = X .


eq or(X, false) = X .
```

Příklad 2: Logická hodnota v Maude

Příklad úpravy (redukce) výrazů

```
red and(X,true) .
red and(true, X) .
red and(and(X,true),and(X,true)) .
red equ(not(X),not(X)) .
red equ(X, X).
```

Maude - instalace

http://maude.cs.uiuc.edu/ Domovská stránka

manuály, src, distribuce pro linux, …

http://maude.cs.uiuc.edu/download/windows.html

src a postup překladu pod cygwin a windows

nutnost, pokud současně cygwin

http://moment.dsic.upv.es/

maude pro windows včetně podmnožiny cygwinu

pokud nemáte cygwin (kolize .dll)

Maude – spuštění

C:\cygwin\usr\local\bin\maude.exe -interactive -no-prelude

-interactive *lze přerušit Ctrl-C*

-no-prelude *bez základních modulů (BOOL, NAT,…)*

Oddělovačem je MEZERA !

všechny příkazy končí TEČKOU (kromě fmod...endfm)

TAB nepoužívat v souborech

– vypisuje soubory vyhovující napsanému začátku

Maude – zápis ADT

ADT je *functional modul*

fmod XXX is

sort, sorts	druhy	… signatura
op	operace	
var, vars	proměnné	… pro zápis axiomů
eg	axiomy (rovnosti)	… semantics

endfm

Maude – atributy operátorů

--- [ctor] je konstruktor

- nezjednodušuje se – je vidět pri obarvování

```
op and : Bool Bool -> Bool [ctor] .
```

--- [iter] umožní zápis s_^5(0) což znamená s s s s s 0

```
op s : Nat -> Nat [ctor iter]
```

--- [comm] komutativní

--- [assoc] asociativni

```
op and : Bool Bool -> Bool  [comm assoc] .
```

Maude - operátory

op and : Nat Nat -> Nat . Prefix

výraz: and(X Y) .

op and(_,_) : Bool Bool -> Bool . Prefix jako v jaz. Clear

výraz: and(X Y) .

op _and_ : Nat Nat -> Nat . mixfix

výraz: X and Y .

Maude – příkazy

in <soubor>	vložení souboru (load)
set trace on .	výpis kroků při redukci
set trace whole on .	včetně výrazu před a po redukci
set print color on .	*barvi symboly dle významu*

(zelený čeká na redukci, červený nejde redukovat)

red <výraz> .	redukce zadaného výrazu
quit .	ukončení maude

show vars . *show eqs .*

show sorts . *show module <JMENO> .*

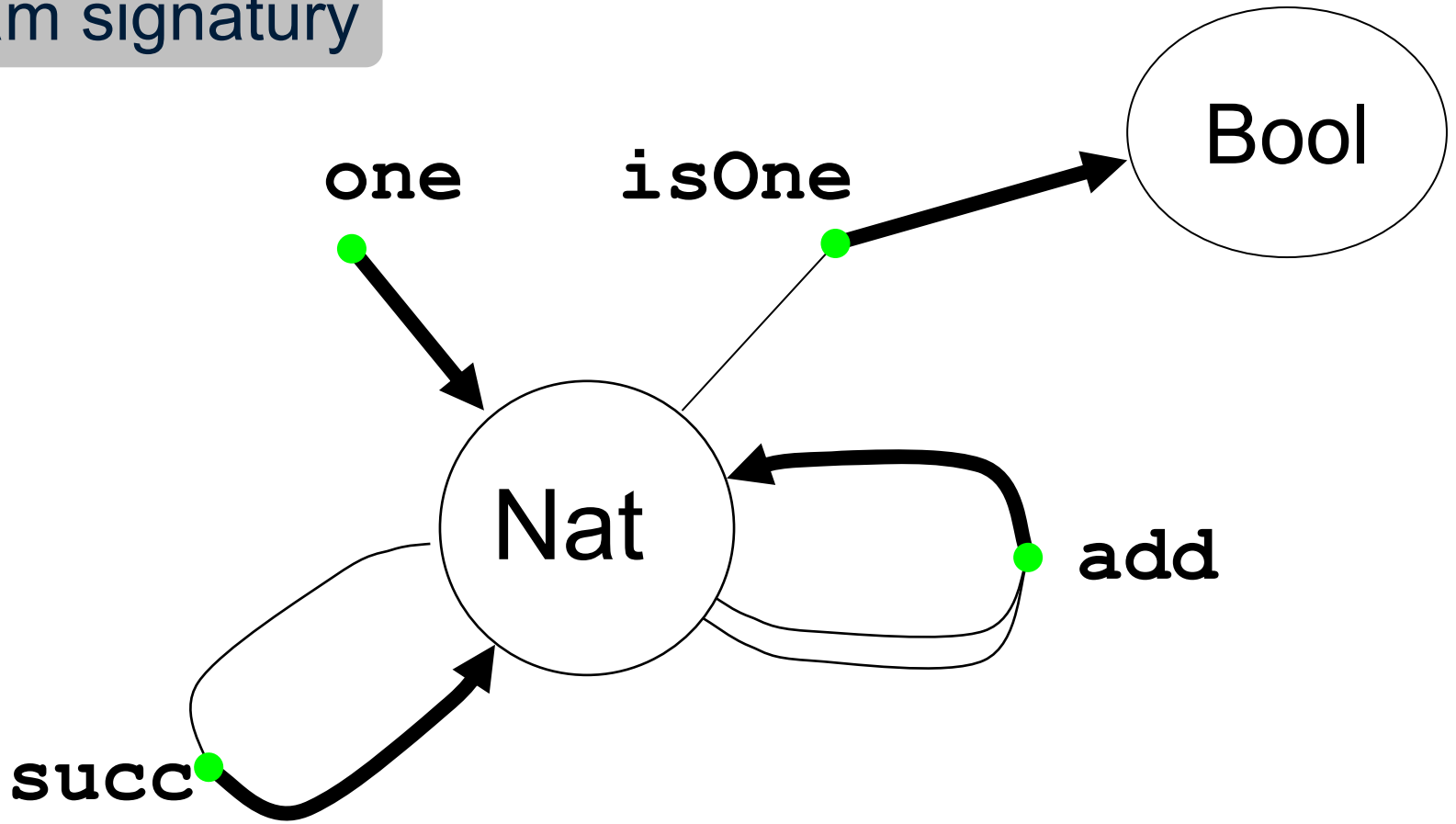
Maude - debug

Např. po Ctrl-C

step .	Krok redukce
where .	Aktuální stav výrazu
abort .	Konec ladění
resume .	Pokračování

Příklad 3: ADT přirozená čísla

Diagram signatury



Příklad 3: ADT přirozená čísla

Sémantika pomocí axiomů

```
var x,y: Nat
isOne(one)       = true     (1)
isOne(succ(x))   = false    (2)
isOne(add(x,y))= false      (3)

add(x, one)      = succ(x)            (4)
add(x,y)         = add(y,x)           (5)
add(succ(x),y)   = succ(add(x,y))     (6)
```

Příklad 3: ADT přirozená čísla

Příklad úpravy výrazu

Příklad:

```
add(succ(one), succ(succ(one)) )=(6)  … (2+3)
succ(add(one, succ(succ(one))) )=(5)  … 1+(1+3)
succ(add(succ(succ(one)), one) )=(6)  … 1+(3+1)
succ(succ(add(succ(one), one) ))=(6)  … 1+1+(2+1)
succ(succ(succ(add(one, one) )))=(4)  … 1+1+1+(1+1)
succ(succ(succ(succ(one) )))          … 1+1+1+1+1
```

Příklad:

```
isOne(add(succ(one),succ(succ(one))))=(3) false
```

Proč i sémantika a ne jen syntaxe?

Příklad: Interval {1, 2, 3, 4, 5, ..., 11, 12}

a) month_in_year

b) natural_number_from_interval_1_to_12

ALE!

Operace *successor* (12)

a) 1 pro month_in_year

b) nedefinována pro number_from_interval

Datové struktury

Datová struktura

= Realizace (implementace) abstraktního datového typu

Např. realizace A pomocí B

1. Zvolí se *datový typ B* kterým se ADT A implementuje …. [int / interval 1..12]
2. Zvolí se *reprezentace objektů* typu A pomocí objektů typu B …. [1..January, 2..February,…]
3. *Operace nad A* se vyjádří *pomocí operací nad B*

int succ(int v) { if(v<n) v++; else v=1; return v }

Shrnutí

Abstraktní datový typ (ADT)

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

signatura = názvy druhů (argumentů+výsledku) a názvy operací,

axiomy = popis vlastností operací

Datová struktura

= Realizace (implementace) abstraktního datového typu

Datový typ

= datová struktura zabudovaná v konkrétním jazyce

Abstraktní datové typy

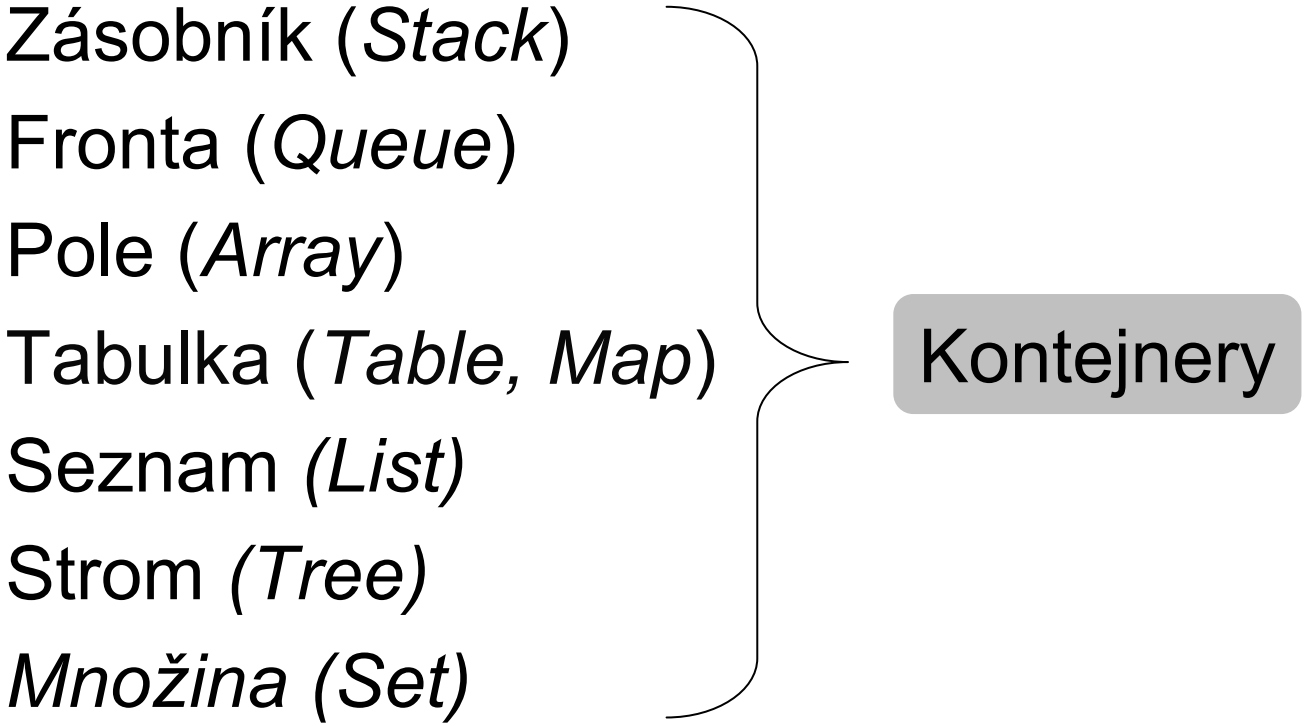
Typické operace

konstruktory ze zadaných parametrů sestaví platnou vnitřní reprezentaci (konstanty)

selektory vrací hodnoty složek ADT

modifikátory mění vnitřní stav ADT

Základní abstraktní datové typy (ADT)



Základní abstraktní datové typy (ADT)

Kontejner (kolekce)

= ADT na organizované skladování jiných objektů podle určitých pravidel

(= po implementaci třída, datový typ)

Sekvenční

Zásobník (*Stack*)

Fronta (*Queue*)

Pole (*Array*)

Seznam *(List)*

Asociativní

Tabulka (*Table, Map*)

Množina (Set)

Základní abstraktní datové typy (ADT)

Kontejner (kolekce)

Očekávané operace

- Vytvoření prázdného kontejneru (konstruktor, *init*)
- Zjištění počtu uložených objektů (*size*)
- Přístup k prvkům kontejneru (*read, top, front*,…)
  (často implementován pomocí *iterátorů*)
- Vložení objektu do kontejneru (*insert*)
- Odstranění objektu z kontejneru (*delete, pop,* …)
- Vymazání všech uložených objektů (*clear*)

(Abstraktní) datové typy

Počet a vzájemné uspořádání složek

- statické = nemění se
- dynamické = mění se

Typ komponent

- homogenní = všechny stejného typu
- nehomogenní = různého typu

Existuje bezprostřední následník

- lineární = existuje [např. pole, seznam,…]
- nelineární = neexistuje [strom, tabulka,…]

Abstraktní datové typy

Zásobník (*Stack*)

Fronta (*Queue*)

Pole (*Array*)

Tabulka (*Table*)

Seznam *(List)*

Strom *(Tree)*

Množina (Set)

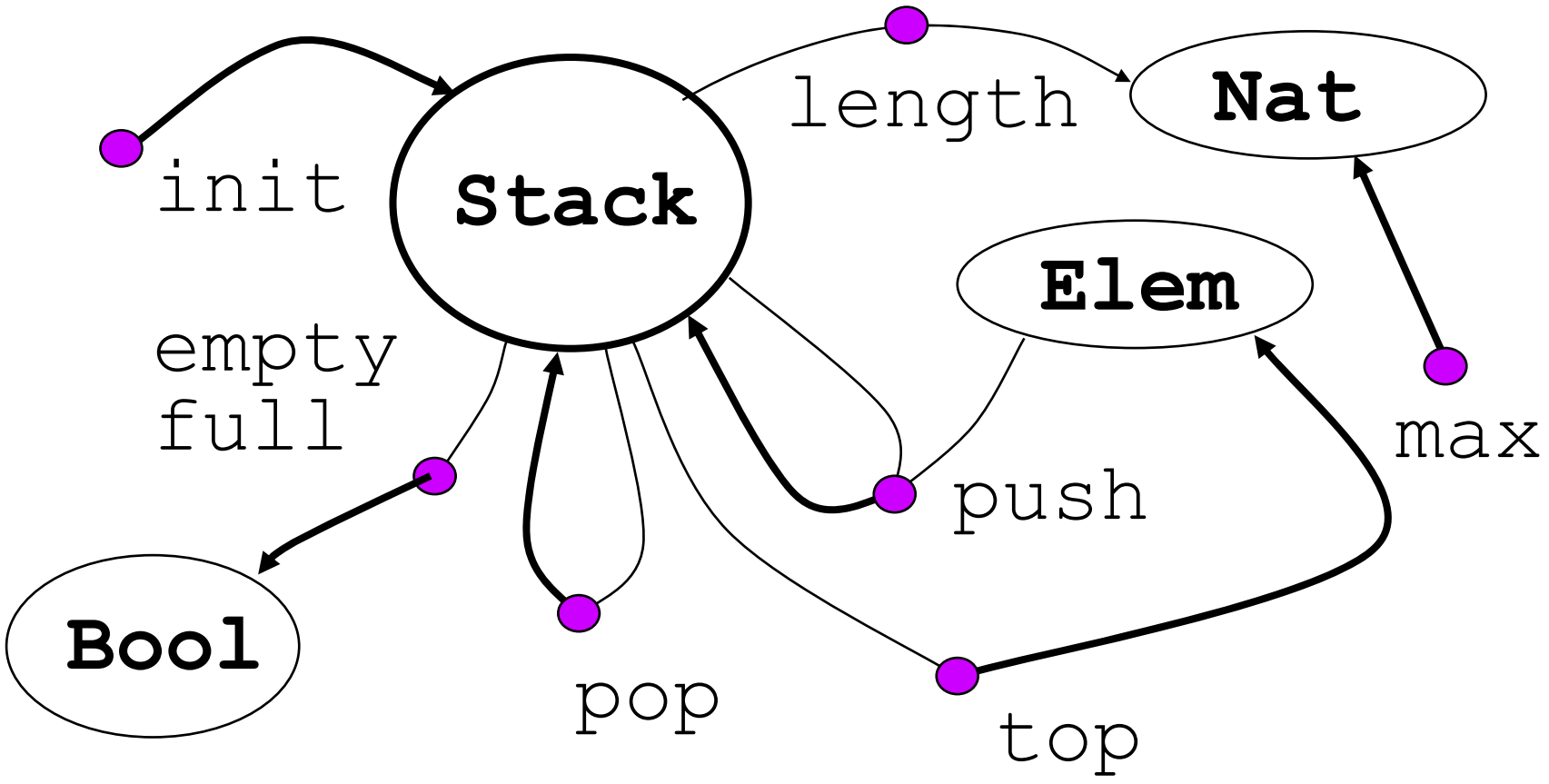
Zásobník (Stack)

Kdy?

- odložení informace, výběr v opačném pořadí (návrat z procedury, uzlové body cesty, náboje v pistoli,...)
- LIFO = *Last-in, First-out*

  „poslední tam, první ven“
- přístup pouze k prvku *na vrcholu (top)*
- vkládání pouze na vrchol (*top*)
- *homogenní, lineární, dynamický*

Stack (Zásobník)



Stack (Zásobník)

Operations:

```
init: -> Stack
empty(_): Stack -> Bool
push(_,_): Elem, Stack -> Stack
top(_): Stack -> Elem
pop(_): Stack -> Stack


length(_): Stack -> Nat
max: -> Nat
full(_): Stack -> Bool ...omezen
```

Stack (Zásobník)

```
empty( init ) = true
empty( push( e, s )) = false

top( init ) = error_elem
top( push( e, s )) = e

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

Stack (Zásobník)

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

Ex:

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

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

Need for more axioms!

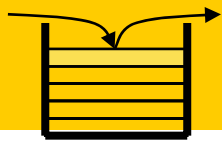
Stack (Zásobník)

```
3. top( init() ) = error_elem()
4. top( push( e, s )) = e
```

Ex:

```
top( push("X", push("Y", pop( push( "A", init()
  ))))) = "X"
top( init() ) = error_elem
top( pop( push( "A", init() ))) = ???
                          Need for more axioms!
```

Stack (Zásobník)



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

Ex:

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

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

Stack (Zásobník) – ext.

```
7. length( init ) = 0
8. length( push( e, s )) =
       succ( length( s ))
```

Ex:

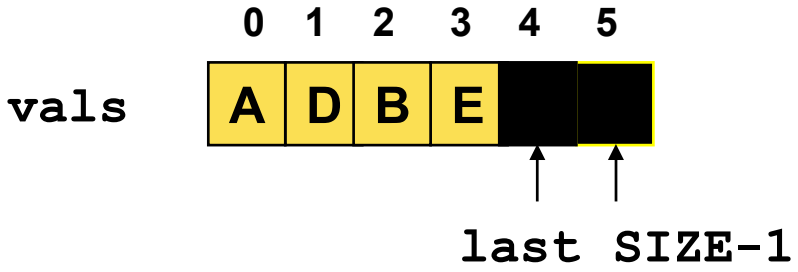
```
length( push("A", init() ))
  = succ( length( init() ))
  = succ( 0 )
  = 1
```

Stack (Zásobník) – limit.

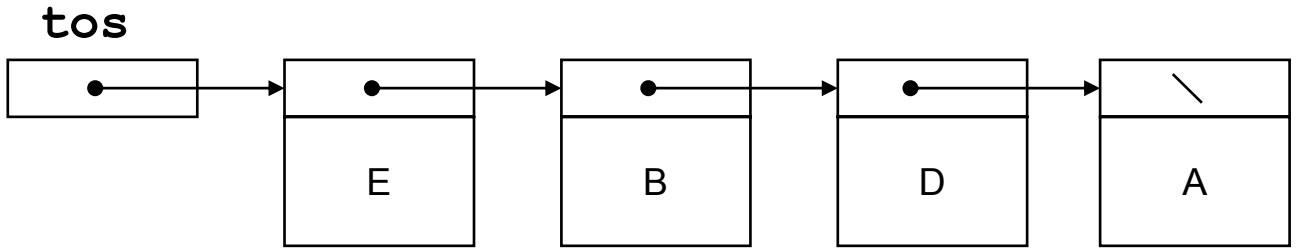
```
 9. full( s ) =  equ( length( s ), max )
10. push( e, s ) = if full( s ) then error_stack
```

Stack - implementation

In Array



In Dynamic Memory



Stack – in Array

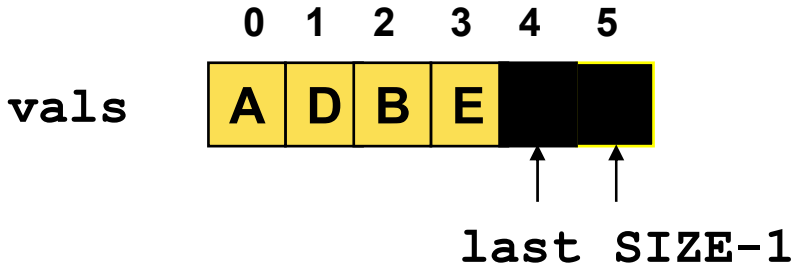
```
class stack
{
  private:
  elem vals[SIZE]; // array of elements
  int last;        // index behind (place for next item)

  public:
  void init( void );

  elem top( void );
  void pop( void );
  void push( elem e );

  void error( const char errorText[] );

  stack(void);     // just calls init()
  ~stack(void);
};
```



Stack – in Array

```
void stack::init(void)      // init: -> Stack
{     last = 0; }
```

```
elem stack::top(void){      // top(_): Stack -> Elem
     if( last > 0 )
         return vals[last-1];
     else
         error("Stack is empty during top()");
     return 0;
}
```

```
void stack::pop(void){      // pop(_): Stack -> Stack
     if( last > 0 )
         last--;
     else
         error("Stack is empty during pop()");
}
```

Stack – in Array

```
void stack::push( elem e )//push(_,_):Elem,Stack -> Stack
{
     if( last < SIZE )
          vals[ last ++ ] = e;

}
```

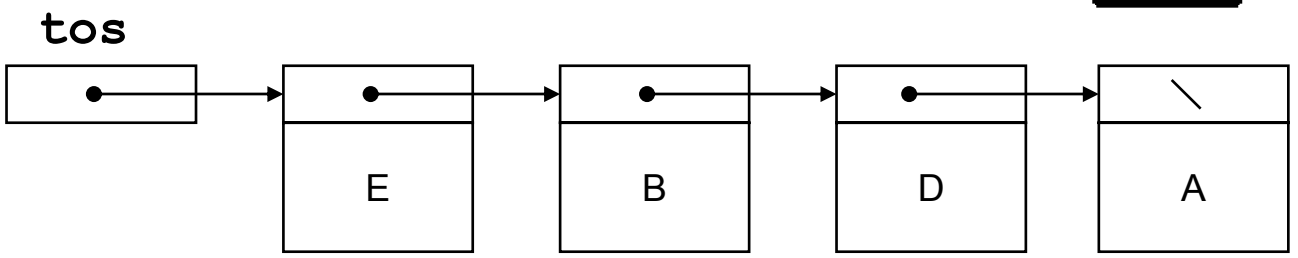
```
bool stack::empty(void)     // empty(_): Stack -> Bool
{    if( last == 0 ) return true;
     else return false;
}
```

Stack – Usage

```
#include "stack.h"
//#include "stackDyn.h"

void main( void )
{
     stack s;                              // OUTPUT:
     cout << s.empty() << endl;            // 1 (true)
     s.push( 2 );
     s.push( 3 );
     cout << s.empty() << endl;            // 0 (false)
     cout << s.top() << endl;              // 3
     s.pop();
     cout << s.top() << endl;              // 2
     s.pop();
     cout << s.top() << endl;              // Stack is empty
     during top
     cout << s.empty() << endl;            // 1 (true)
}
```

Stack – in Dynamic Mem.




```
class stack
{
  private:
  Item * tos;  // head of the list of elements,
               // tos = Top Of Stack

  public:
  void init( void );

  elem top( void );
  void pop( void );
  void push( elem e );

  void error( const char errorText[] );

  stack(void);        // just calls init()
  ~stack(void);
};
```

Stack – in Dynamic Mem.

```
void stack::init(void)    // init: -> Stack
{     tos = NULL; }
```

```
elem stack::top(void){    // top(_): Stack -> Elem
      if( tos != NULL )
          return tos->val;
      else
          error( "Stack is empty during top()" );
      return 0;
}
```

```
void stack::pop(void){    // pop(_): Stack -> Stack
      if(  tos != NULL ) {
          Item* tmp = tos;
          tos = tos ->next;
          delete( tmp );
      }
      else
          error( "Stack is empty during pop()" );
}
```

Abstraktní datové typy

Zásobník (*Stack*)

Fronta (*Queue*)

Pole (*Array*)

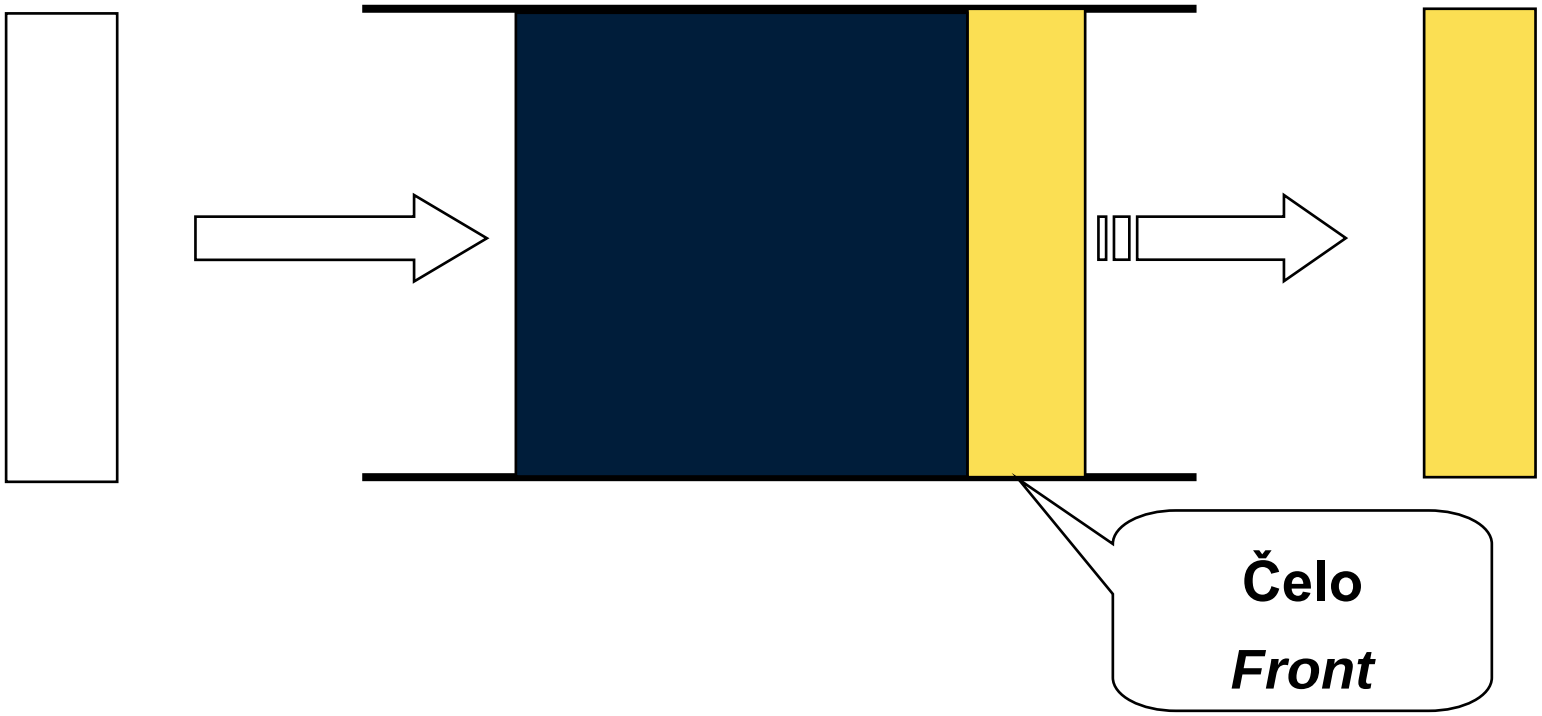
Tabulka (*Table*)

Seznam *(List)*

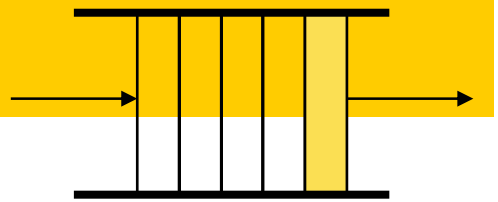
Strom *(Tree)*

Množina (Set)

Queue (Fronta)



Queue (Fronta)



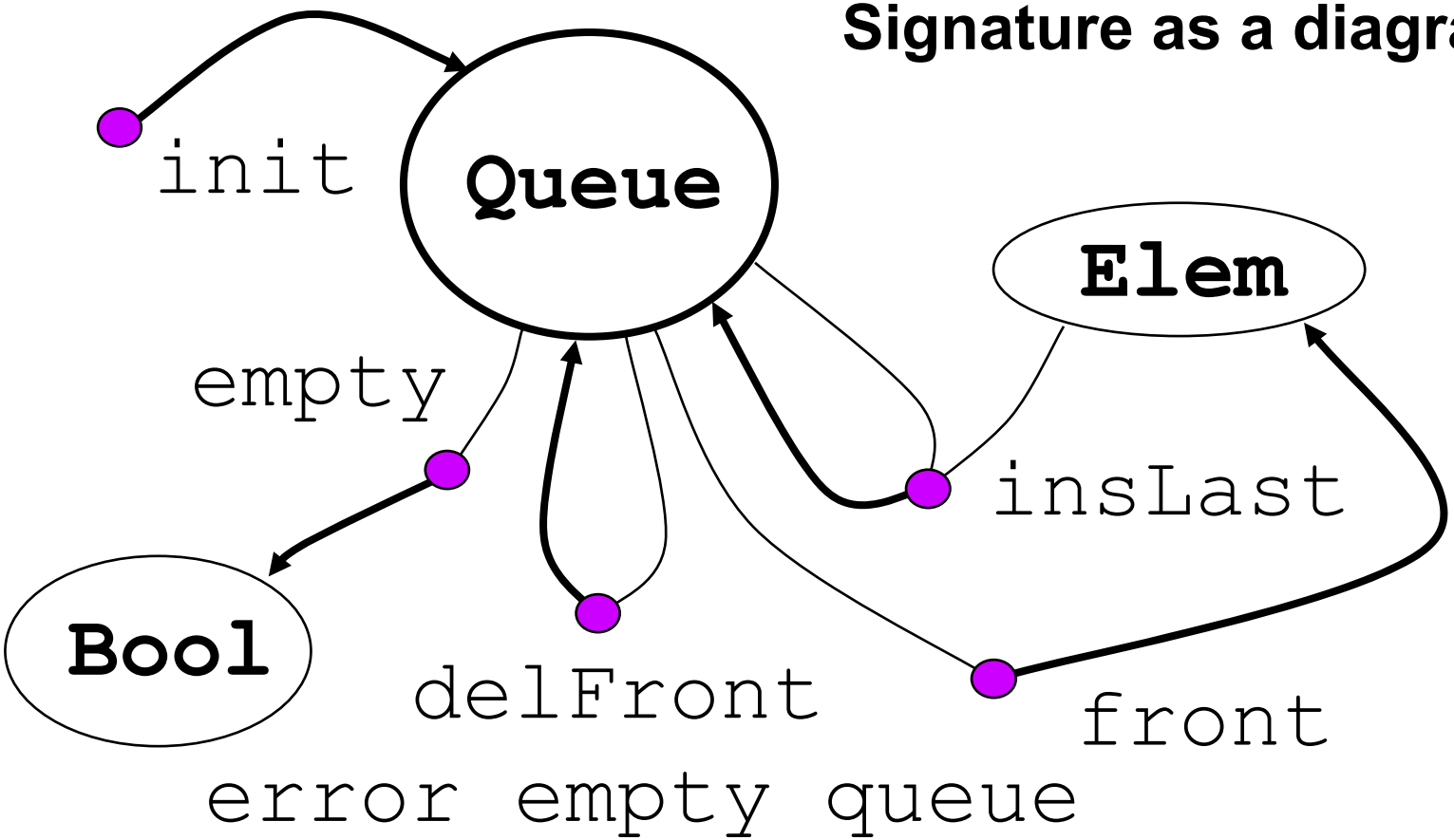
FIFO = *First-in, First-out*

Přístup k začátku (front)

Vkládání na konec (back)

Queue SIGNATURE

Signature as a diagram



Queue SIGNATURE

Signature in symbols

```
init: -> Queue
insLast(_,_): Elem,Queue -> Queue
empty(_): Queue -> Bool
front(_): Queue -> Elem
delFront(_): Queue -> Queue
```

Queue (Fronta)

Operace:

Proměnné: `e`…element, `Q`…queue

`init(), front(Q), delFront(Q), insLast( e, Q )`

Axiomy:

1. **`init()`** vrácí prázdnou

---

2. **`front( insLast( e, init() )) = e`**
3. **`delFront( insLast( e, init() ) ) = init()`**

---

4. **`front( insLast( e, insLast( f, Q ) )) =`**
   `front( insLast( f, Q ) )`

5. **`delFront( insLast( e, insLast( f, Q ) )) =`**
   `insLast( e, delFront( insLast( f, Q ) ))`

Queue (Fronta)

Příklady výrazů

insLast(4, insLast(3, insLast(2, init())))

reprezentuje frontu

front(insLast(4, insLast(3, insLast(2, init()))))

reprezentuje její první prvek (head), … 2

Queue (Fronta)

1. init()

Returns an empty queue

Is informally “empty”

Must not be formally defined

Queue (Fronta)

2. `front( insLast( e, init() )) = e`

= is a “rewrite operation”

axiom **A=B** means instead **A**, write **B**

Ex:

front(insLast(99, init())) means **99**

Queue (Fronta)

3. `delFront( insLast( e, init() )) = init()`

Axioms allow reduction of expressions

Ex.

```
delFront( insLast( 6, init() ))
```

means

```
init()
```

Queue (Fronta)

```
4. front( insLast( e, insLast( f, Q ) )) =
          front( insLast( f, Q ) )
```

Ex:

```
front( insLast( 11, insLast( 5, init() )))
```

means [according to axiom 4]

```
front( insLast( 5, init() ))
```

means [ax. 2]

```
5
```

Queue (Fronta)

5. `delFront( insLast( e, insLast( f, Q ) )) = insLast( e, delFront( insLast( f, Q ) ))`

Ex:

```
delFront( insLast( 8, insLast( 6, init() )))
```

means [according to axiom 5]

```
insLast( 8, delFront( insLast( 6, init() )))
```

means [ax. 3]

```
insLast( 8, init() )
```

Queue (Fronta)

And what to do with:

```
front( init() )       ?
delFront( init() )    ?
```

Behavior is in the queue undefined!

Add error states (in implementation use exceptions)

```
error_empty_queue()
error_empty_elem()
```

```
6. front( init() ) = error_empty_elem()
7. front( error_empty_queue() ) =
                    = error_empty_elem()
```

Ignore error state

```
8. delFront( init() ) = init()
```

Queue (Fronta) complete

`init()` returns a queue

`error_queue()` returns an invalid queue

Variables: `e`…element, `Q`…queue

```
front( init() ) = error_elem()
front( insLast( e, init() )) = e
front( insLast( e, insLast( f, Q ) )) =
           front( insLast( f, Q ) )
front( error_queue() ) = error_elem()

delFront( init() ) = init()
delFront( insLast( e, init() ) ) = init()
delFront( insLast( e, insLast( f, Q ) )) =
            insLast( e, delFront( insLast( f, Q )))
delFront( error_queue()) = error_queue()
```

Queue - extensions

Variables: e…element, Q…queue

```
empty( init() ) = true
empty( insLast( e, Q )) = false

length( init() ) = 0
length( insLast( e, Q ) ) = succ( length( Q ) )
```

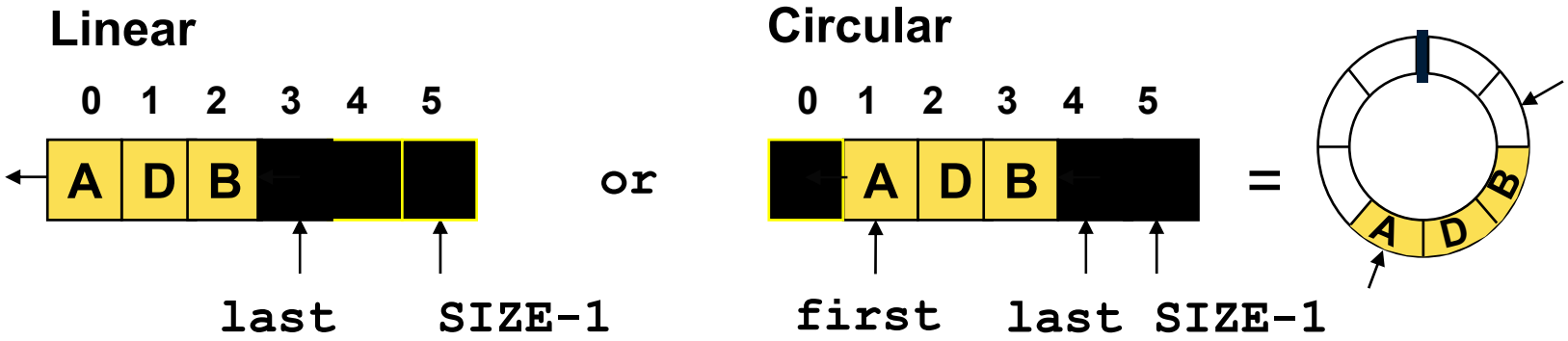
…

Implementation

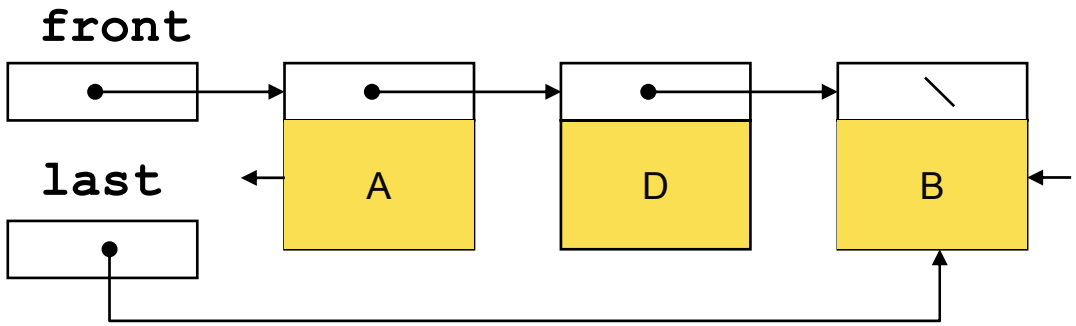
From
Abstract Data Type
to
Data Structure

Queue implementation

In Array



In Dynamic Memory



Example – SPECIFIC Queue

Interface (Syntax):

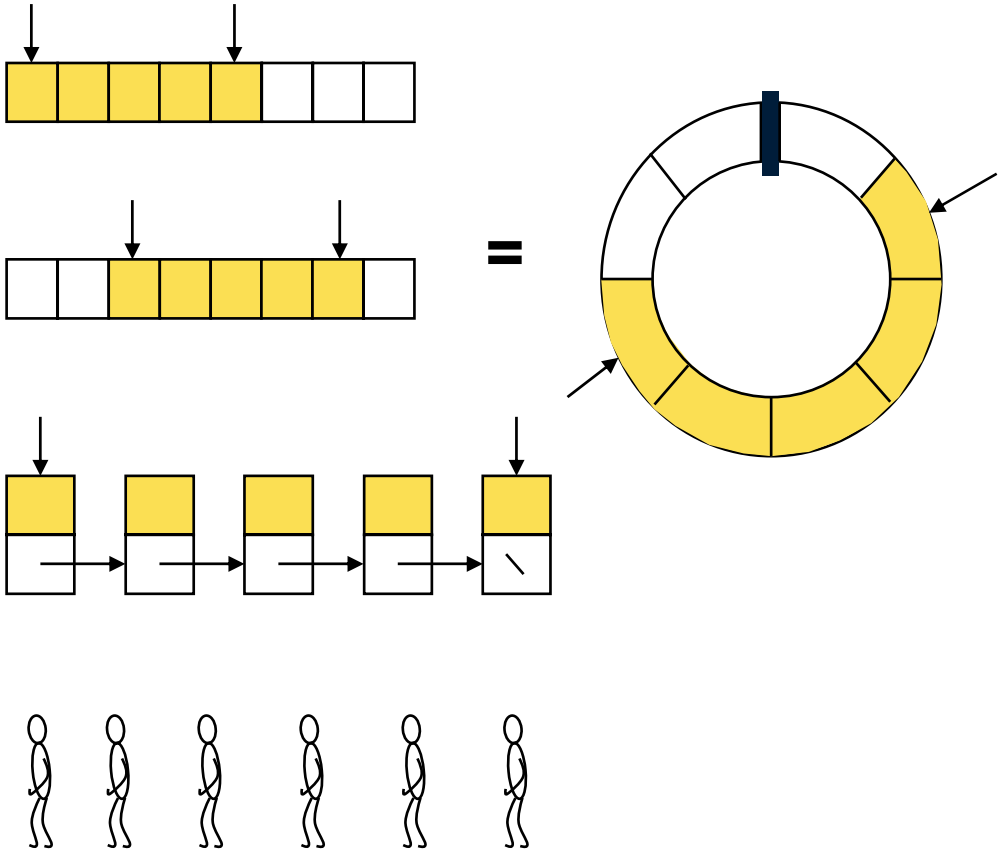
Coupled together with

```
Class Queue
{  int _size;
   Elem elems[MAX_SIZE];
```

array

```
  public:
   Queue  init();
   Elem front();
   Queue delFront();
   Queue insLast( Elem elem );
   bool empty();
   …
}
```

Implementation:



Abstract x Specific Queue

Axioms

```
vars: Elem e; Queue q;
eqns:
   empty( init ) = true
   empty( insLast(e, q))
     = false
```

Implementation

Array, list

```
bool empty() {
  return (_size == 0)
};
```

or

list

```
bool empty() {
  return (_front == null )
};
```

Queue – in Linear Array

```
Elem queue::front(void) {
    if( lastIdx == 0 )
        return error_empty_elem();
    else
        return array[1];
}
```

```
void queue::delFront(void)
{
    if(lastIdx != 0)  //!empty()
    {
        lastIdx --;
        for(int i=0; i<lastIdx; i++)
            array[i] = array[i+1];
    }
}
```

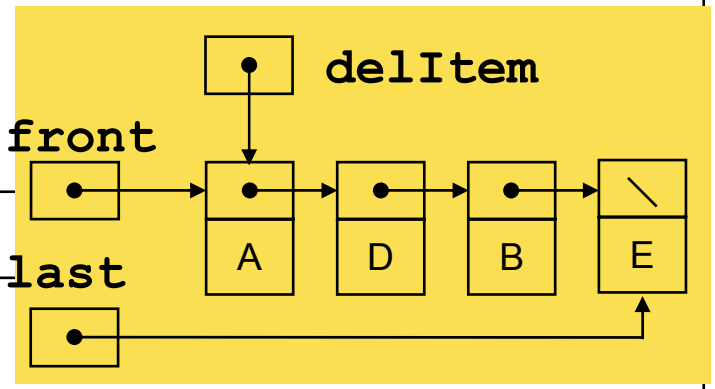
Queue – in Circular Array

```
Elem front() {
  if(_frontIdx == _lastIdx )
     return error_empty_elem();
  else
    return( array[_frontIdx] );
};
```

```
void queue::delFront(void)
{
     if(frontIdx != _lastIdx )
     {
         frontIdx =(frontIdx + 1) % array.size();
     }
}
```

Queue – in Dynamic Mem.

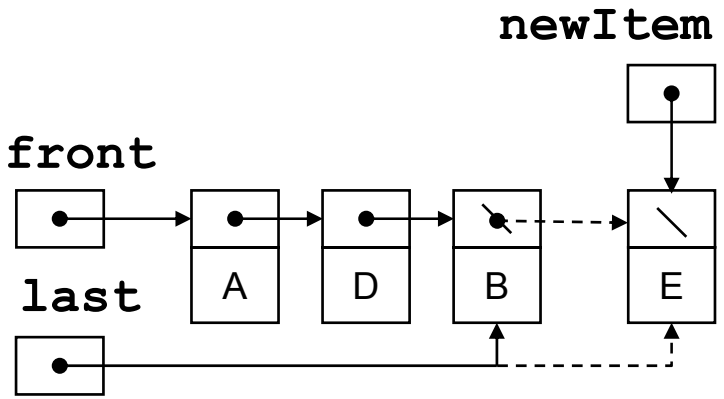
```
Elem queue ::front(void) {
     if( last == null )
         return error_empty_elem();
     else
         return front->val;
}
```




```
void queue::delFront(void)
{
     if( front != null) { //!empty()
         Item *delItem = front;
          front = front.next;
         delete delItem;
     }
}
```

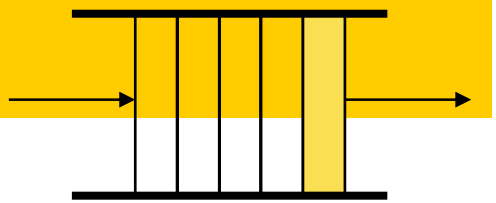
Queue – in Dynamic Mem.

```
void queue::insLast( elem e )
{
     Item *newItem = new Item;
     newItem->val = e;
     newItem->next = null;
     last.next = newItem;
     last = newItem;
}
```




```
bool queue ::empty(void) // empty(): Queue ->Bool
{
     if( last == NULL  ) return true;
     else return false;
}
```

Fronta – Shrnutí



- úlohy *hromadné obsluhy*
- FIFO = *First-in, First-out*

  kdo dřív přijde, ten dřív mele
- *přístup pouze k prvnímu prvku (front)*
- vkládání pouze na konec (*back*)
- *homogenní, lineární, dynamická*

Abstraktní datové typy

Zásobník (*Stack*)

Fronta (*Queue*)

Pole (*Array*)

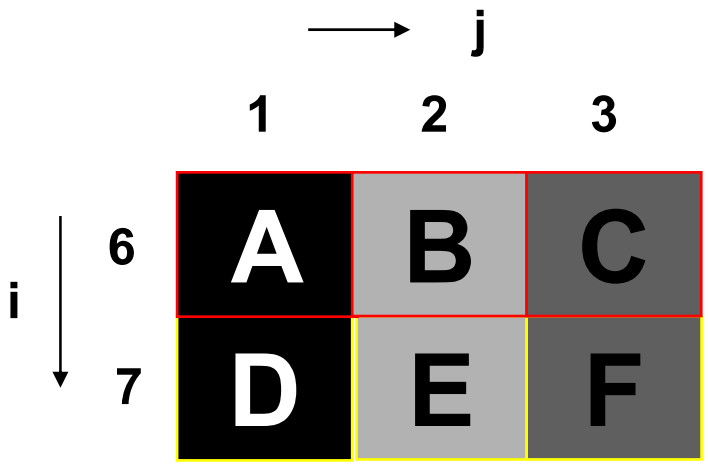
Tabulka (*Table*)

Seznam *(List)*

Strom *(Tree)*

Množina (Set)

Pole (array)

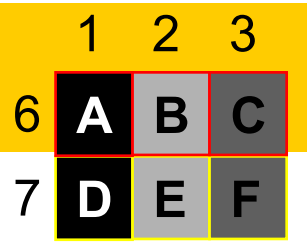


Array a of 2x3 elements

- Row index $i \in \{6,7\}$
- Column index $j \in \{1, 2, 3\}$

a[6,3] -> C

Pole (array)

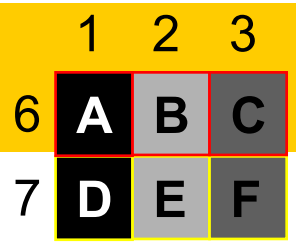


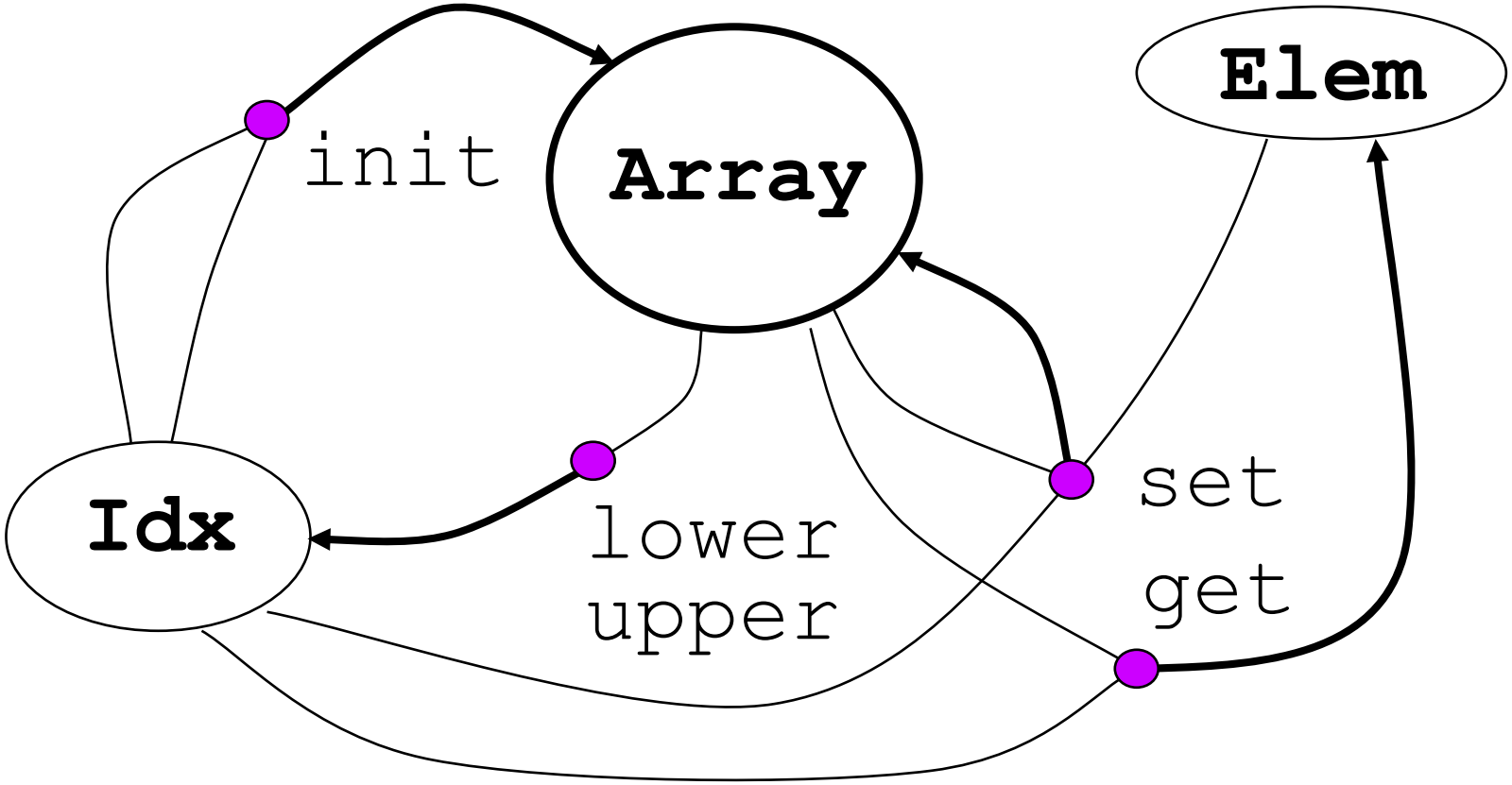
Kdy?

- prvky *stejného* typu - *homogenní*
- všechny prvky *současně* v paměti
- rychlý *náhodný přístup* (*random-access*)
- *známý počet* prvků - *statické*
- indexy uspořádány - *lineární*

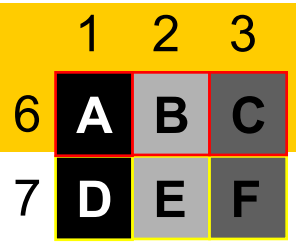
- dán počet dimenzí (*n) a* meze indexů
- přístup k prvkům - mapovací funkce

Pole (array)





Pole (array)



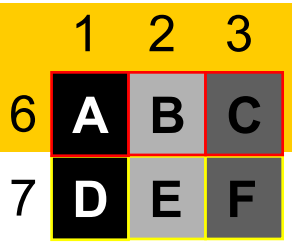
Operations

```
init(_,_): Idx, Idx -> array
upper(_): Array -> Idx
lower(_): Array -> Idx
set(_,_,_): Array,Idx,Elem->Array
get(_,_): Array, Idx -> Elem
```

1-D (one dimensional)

n-D -> `Elem` ~ (n-1)D `Array`

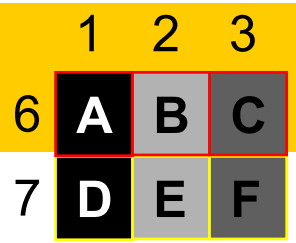
Pole (array)




```
lower( init(m,n) ) = m
lower( set(a,i,e) ) = lower( a )
upper( init(m,n) ) = n
upper( set(a,i,e) ) = upper( a )

set( a, i, e ) =
  if( or(   lt(i,lower(a)),
            lt(upper(a),i)
        )) then error_array
  else set( a, i, e )
```

Pole (array)



$$get(\ init(m,n),\ i\ ) = error\_elem$$

$$get(\ set(\ a,\ i_1,\ e\ )\ ,\ i_2\ ) =$$

$$= if(\ equ(i_1,\ i_2))\ then\ e$$

$$else\ get(\ a,\ i_2\ )$$

Praxe:

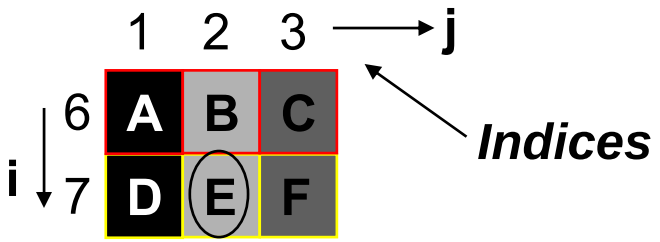
array[i] = e znamená set(array,i,e)

e = array[i] znamená e = get(array,i)

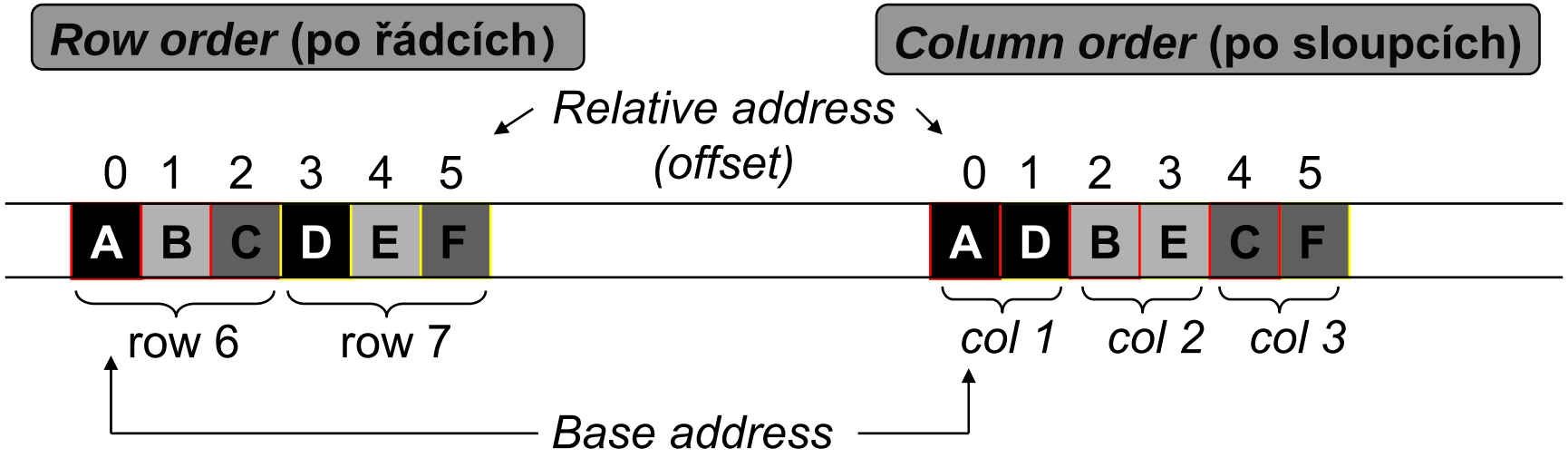
Array (Pole) – mapping function

Logical structure

```
Array a[6..7, 1..3]
Ex: a[7,2]->E
```



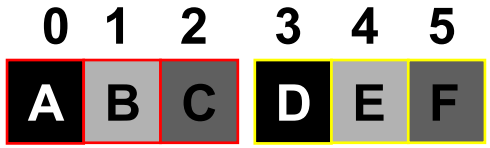
Physical location in memory




```
map(7,2) -> (base + 4)
```

```
map(7,2) -> (base + 3)
```

Array (Pole) – mapping function



***row order* (po řádcích) 2D**

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

Row offset (# rows to skip)

row length = Number of columns

Base address

Column offset

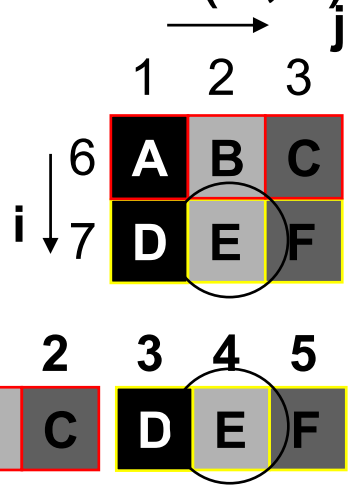
Relative address

$$map(i,j) = a(0,0) + (i*n_j + j)$$ … for indices from (0,0)

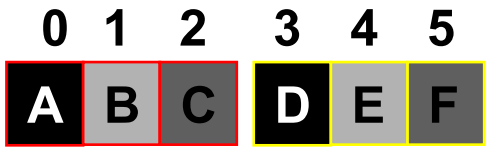
Ex: map(7,2) = a(6,1) + (i-6)*3 + (j-1)

map(7,2) = a(6,1) + (7-6)*3 + (2-1)

= a(6,1) + 4



Array (Pole) – mapping function



***row order* (po řádcích) 3D**

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

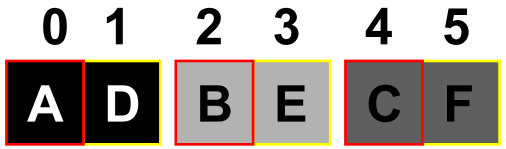
Base address

Relative address (element offset)

$$map(i,j,k) = a(0,0,0) + (i*n_j + j) * n_k + k$$

… for indices from (0,0)

Array (Pole) – mapping function



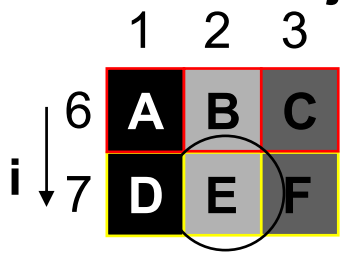
***column order* (po sloupcích) 2D**

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

Base address

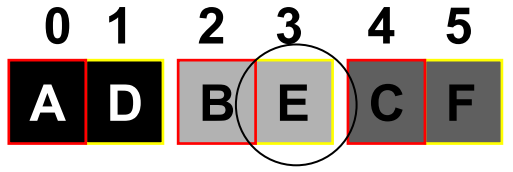
Relative address (element¨offset)

$map(i,j) = a(0,0) + j * n_i + i$ … for indices from (0,0)

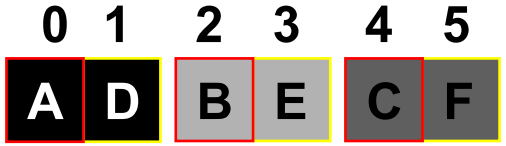


Ex: map(7,2) = a(6,1) + (7-6) + (2-1) * 2

= a(6,1) + 3



Array (Pole) – mapping function



***column order* (po sloupcích) 3D**

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

Base address

Relative address

$$map(i,j,k) = a(0,0,0) + (k*n_j + j) * n_i + i$$

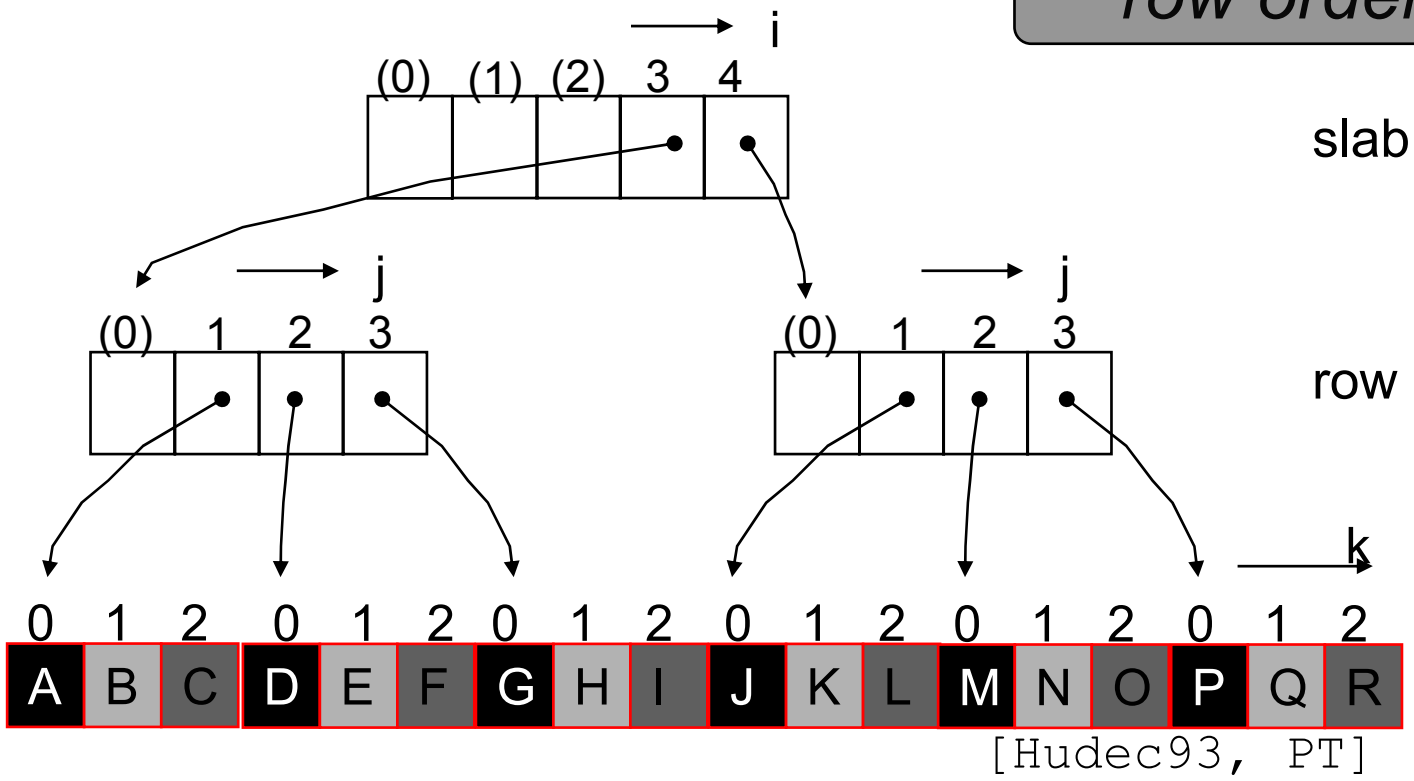
… for indices from (0,0)

Array – speedup by Iliff's vectors

Mapping functions need multiplications -> change them to additions

`A: array [3..4, 1..3, 0..2]`

row order 3D



map [i,j,k] = base(0,0,0) + k + row[j + slab[i]]

Abstraktní datové typy

Zásobník (*Stack*)

Fronta (*Queue*)

Pole (*Array*)

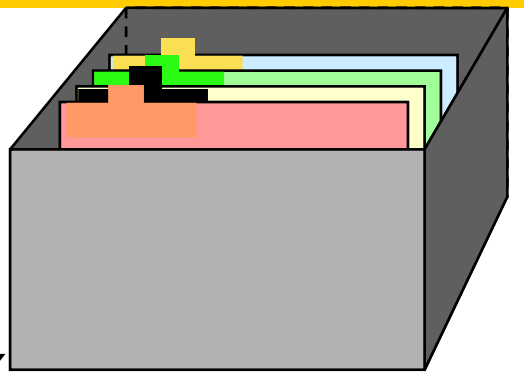
Tabulka (*Table*)

Seznam *(List)*

Strom *(Tree)*

Množina (Set)

Vyhledávací Tabulka (Look-up Table)



Kartotéka, asociativní paměť,
převod mezi kódy, četnost slov,...

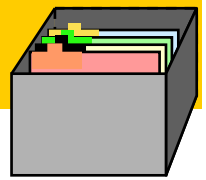
Homogenní, dynamická (nejen) a nelineární

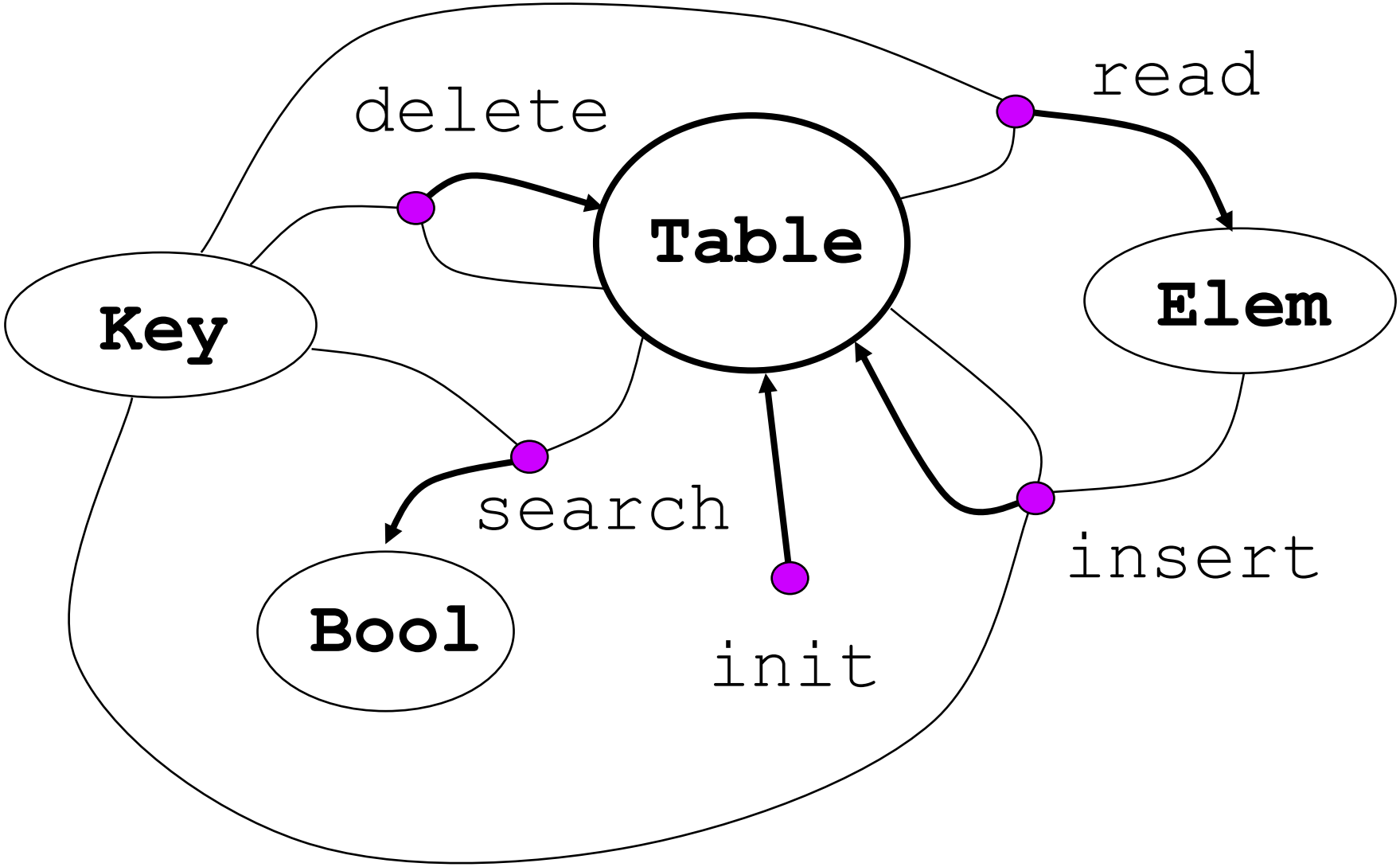
Obsahuje

- *položky*
- jednoznačně identifikované *klíčem* (podle klíče se položky vyhledávají)
- klíč jen 1x (je jedinečný)

Př.: seznam hospod, klíčem je jejich jméno.

Tabulka (Look-up Table)



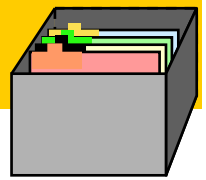


Tabulka (Look-up Table)

Operace:

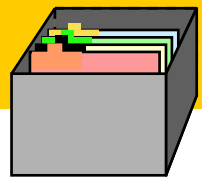
```
init: -> Table
insert(_,_,_): Key, Elem, Table
                -> Table
read(_,_): Key, Table -> Elem
delete(_,_): Key, Table -> Table
search(_,_): Key, Table -> Bool
```

Tabulka (Look-up Table)



```
search( k, init ) = false
search( k1, insert(k2,e,t)) =
  if( equ(k1, k2) )
        then true
        else search( k1, t )
```

Tabulka (Look-up Table)

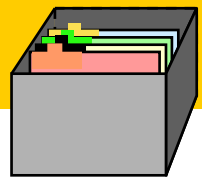


```
delete( k, init ) = init
delete( k1, insert(k2,e,t)) =
 if( equ(k1, k2) )
  then delete( k1, t )
  else insert(k2,e,delete(k1, t))
```

Tabulka (Look-up Table)

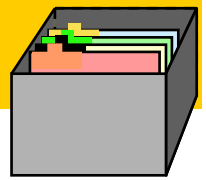
```
read( k, init ) = error_elem
read( k1, insert(k2,e,t)) =
 if( equ(k1, k2) )
  then e
  else read( k1, t )
```

Tabulka (Look-up Table)



```
insert(k1,e1, insert(k2,e2,t)) =
  if( equ(k1, k2) )
     then insert(k1,e1, t)
     else insert(k2,e2,insert(k1,e1, t))
```

Tabulka – Implementace



V poli

- pro malé universum klíčů
  - sekvenční vyhledávání … O(n)
  - přímý přístup (klíč = index) … O(1)
    - convertToLatin2(char_kamenický)

```
LUT[char_kamenický]
```

- Pro velké universum klíčů
  - Rozptylovací tabulky (hash)

… průměrně O(1), max. O(n)

```
Viz přednáška číslo 7
```

Abstraktní datové typy

Fronta (*Queue*)

Zásobník (*Stack*)

Pole (*Array*)

Tabulka (*Table*)

Seznam *(List)*

Strom *(Tree)*

Množina (Set)

Prameny

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

Gang of four (Cormen, Leiserson, Rivest, Stein): Introduction to Algorithms, MIT Press, 1990

Code exapmples: 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. 10 February 2005. (accessed 10.2006) Available from:
http://www.nist.gov/dads/HTML/abstractDataType.html

"Abstract data type." Wikipedia, The Free Encyclopedia. 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