Datové struktury a algoritmy

Část 5

Abstraktní datové typy

Petr Felkel

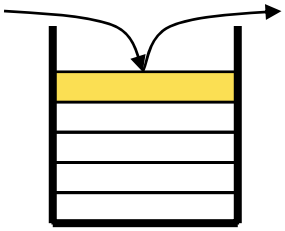
S použitím materiálů [Richta, Honzík, Hudec, Beneš]

05.11. 2007

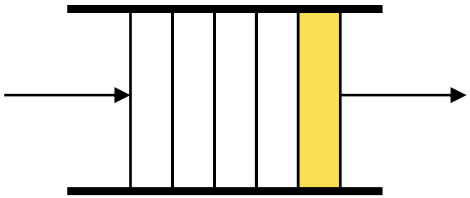
Abstraktní datové typy - opakování

Malé opakování

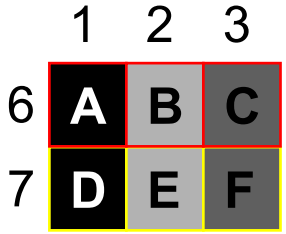
Zásobník



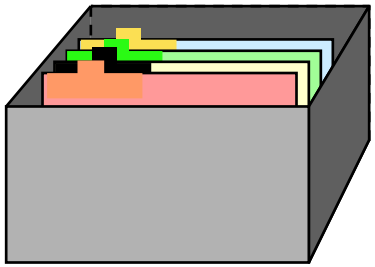
Fronta



Pole



Tabulka



Sekvenční

Asociativní

tos()

front()

index

klíč->hodnota

Implementace v STL

Kontejnery a adaptéry

Sekvenční

vector [] … “nafukovací pole

adaptér:

priority_queue

dqueue … oboustranná fronta

adaptéry:

stack

queue

list … linked list

Asociativní

set …množina bez opakování

multiset … mn.s opakováním

map … pár <klíč-hodnota T>

multimap … <klíč,T,T,T>

Plus iterátory na procházení

Implementace v STL

Použití - vektor

```
#include <vector>
```

... nebo <vector.h> v HP implementaci

```
int main() {
  std::vector<int> v;

  v[2] = 7;
}
```

Nebo:

```
using std::vector;
vector<int> v;
```

Implementace v STL

Použití - fronta

```
#include <queue>
int main() {
  std::queue<int> Q;
  Q.push(3);
  Q.push(2);
  int a = Q.front();
  Q.pop();
  …
}
```

Nebo:

```
using std::queue;
queue<int> Q;
```

Prioritní fronta

Operace má jako fronta, ale front() vrací extrém:

minimum nebo maximum

Implementace: haldou (v poli)

Insert, delete…. O(log n)

Abstraktní datový typ

SYNTAXE

… signatura

= deklarace druhů

- Jména oborů hodnot

\+ deklarace operací

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

SÉMANTIKA

… axiomy

= popis vlastností operací!

Abstraktní datové typy

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

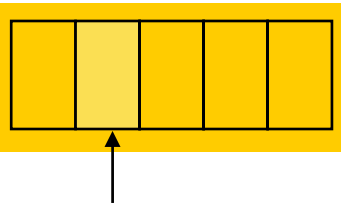
Seznam *(List)*

Strom *(Tree)*

Množina *(Set)*

[Množina s opakováním prvků (multiset) – k zamyšlení]

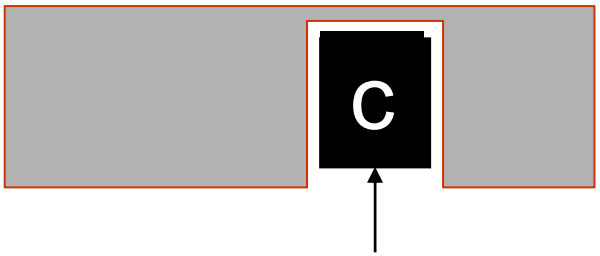
Seznam (List)



Posloupnost údajů

+

Ukazovátko!!!!;



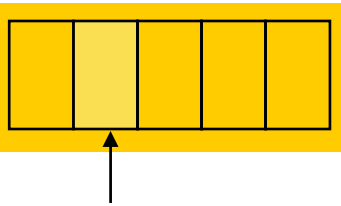
Přidat / zrušit / měnit prvek lze pouze v místě ukazovátka!

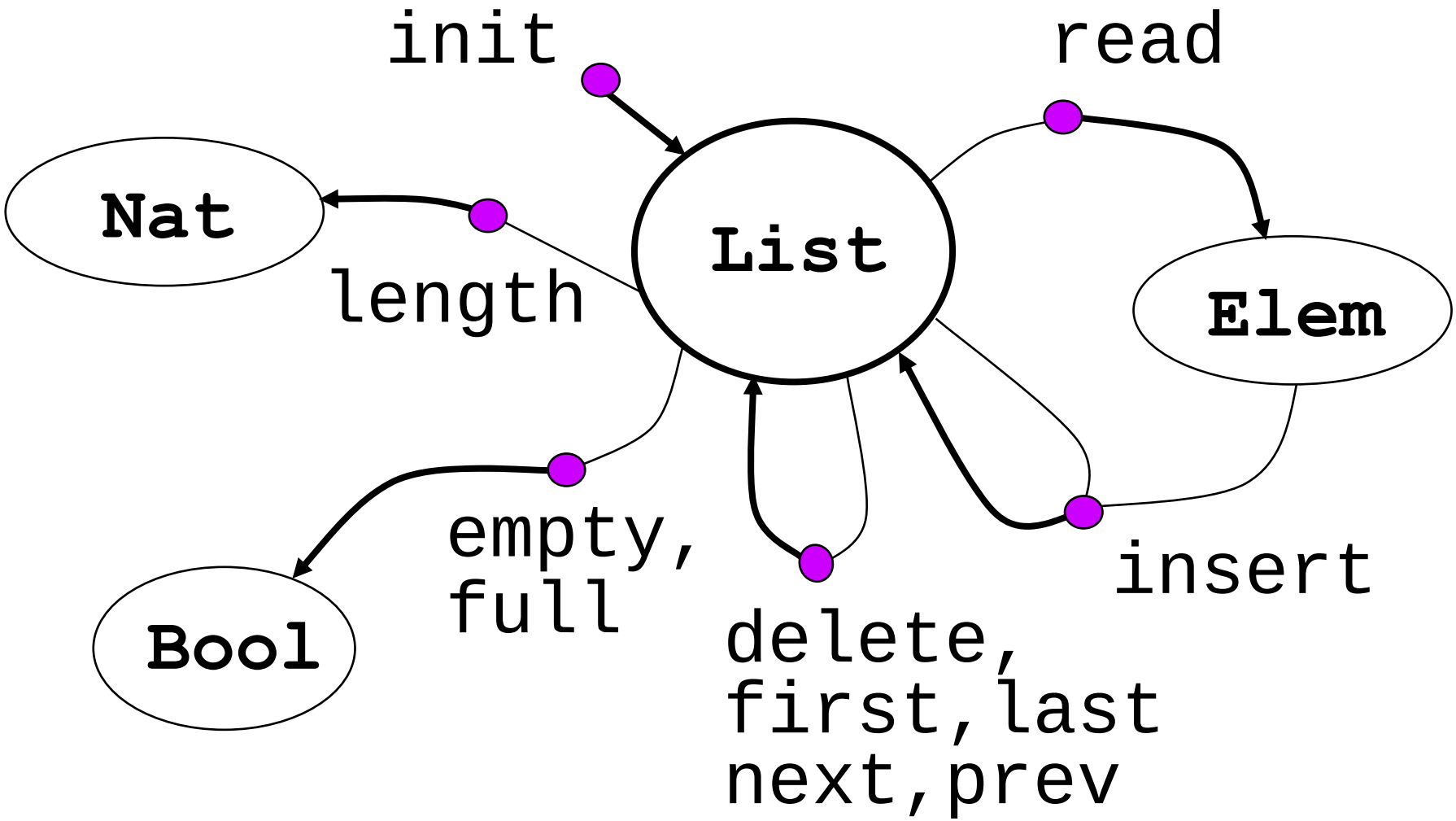
Homogenní, lineární, dynamická

Př: Linked list = seznam v dynamické paměti (STL) O(1)

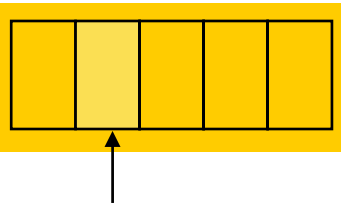
(Ne ArrayList v Javě, který má get(i) se složitostí O(n))

List (Seznam)




init
read
Nat
List
Elem
length
empty,
full
insert
Bool
delete,
first,last
next,prev

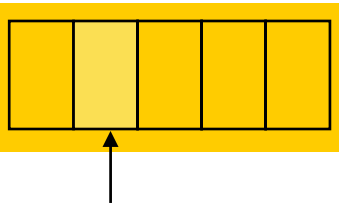
List (Seznam)



Operace ADT seznam

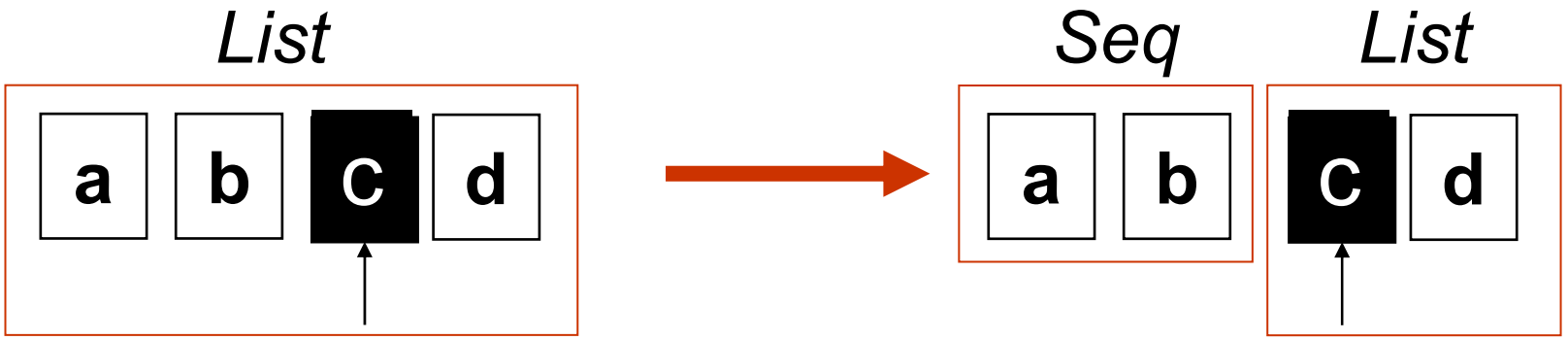
```
init: -> List
insert(_,_): Elem, List -> List
read(_): List -> Elem
delete(_),
first(_), last(_)
next(_), prev(_): List -> List
length(_): List -> Nat
empty(_), full(_)
atbeg(_), atend(_): List -> Bool
```

List (Seznam)



Zavedeme **Pomocný druh** sekvence **Seq**

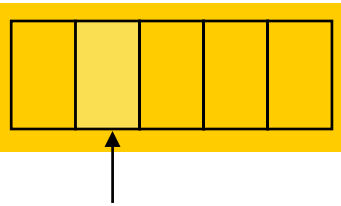
Seq = posloupnost bez ukazovátka



a pomocné operace

- NEJSOU operacemi rozhraní LIST
  (Nejsou „Zvenku“ vidět)
- Použijeme je jen na naše axiomy
- Umožní vyjádřit sémantiku operací

List (Seznam)

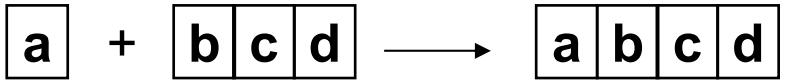


Pomocné operace (nejsou z venku vidět!!!!!):

`new: Seq` ... prázdná posloupnost

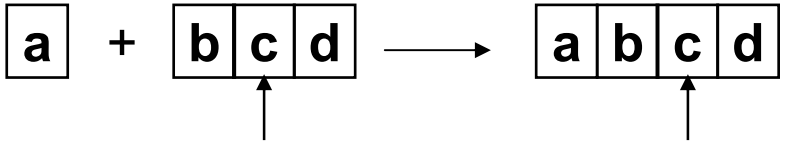
`cons(_,_) : Elem, Seq -> Seq`

… vložení prvku do čela posloupnosti



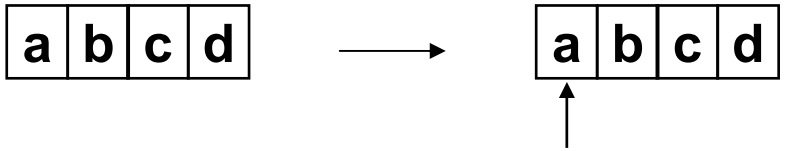
`consL(_,_): Elem, List -> List`

… vložení do čela seznamu bez ohledu na ukazovátko

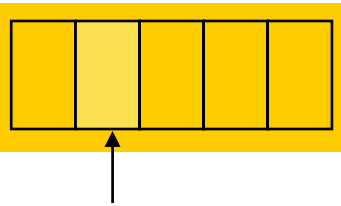


`mark(_): Seq -> List`

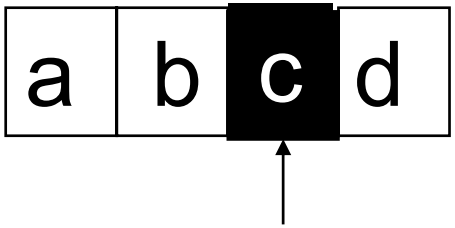
... Přidání ukaz. do čela posloupnosti – změní se na seznam



List (Seznam)

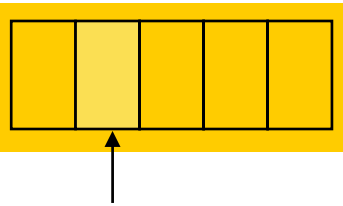


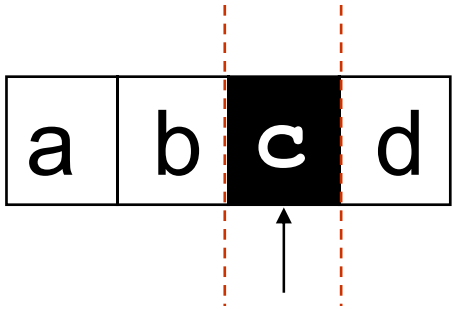
Příklad vnitřní reprezentace:



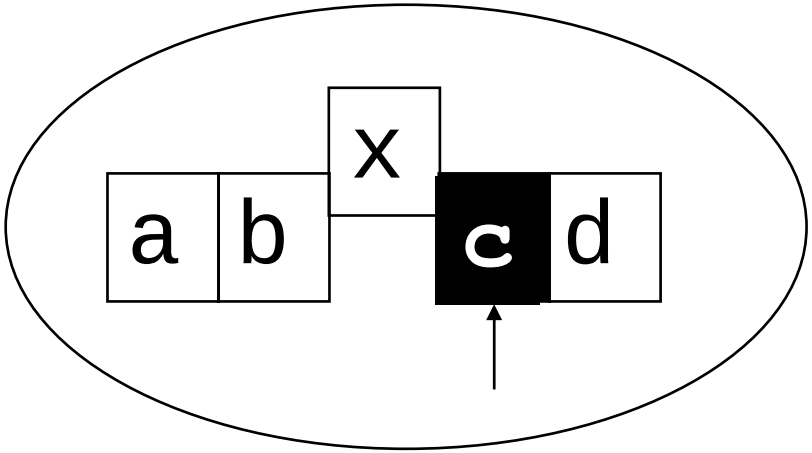

```
consL( a, consL( b,
    mark( cons( c, cons( d, new )))
))
```

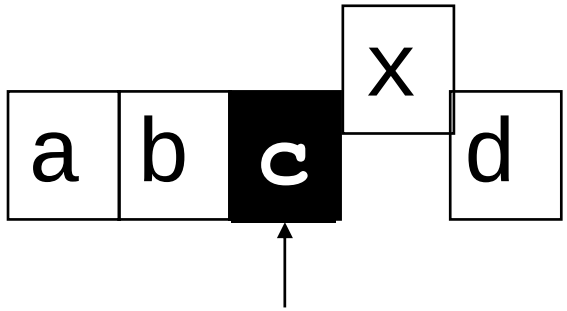
List – insert 1/2





Kam nový prvek vložit?

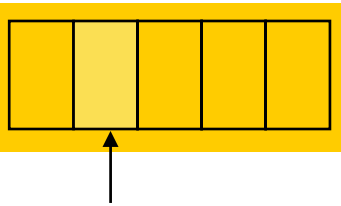


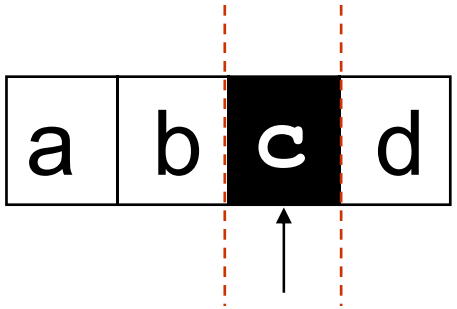


Před aktuální prvek?

Nebo za aktuální?

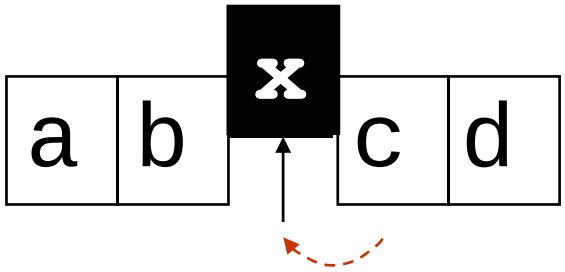
List – insert 2/2

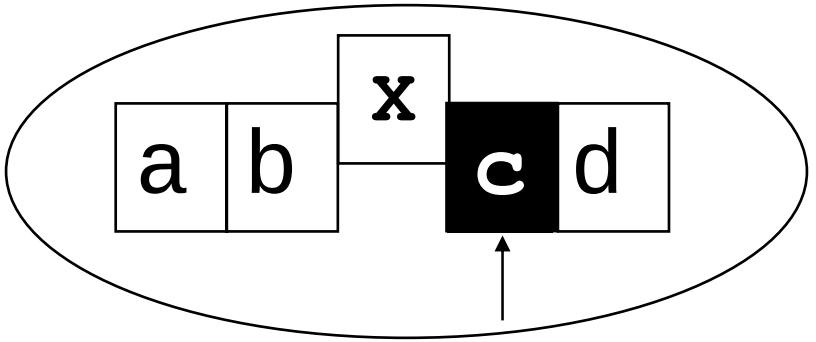




Vkládáme *před aktuální prvek*.

Kam nastavit ukazovátko po vložení?



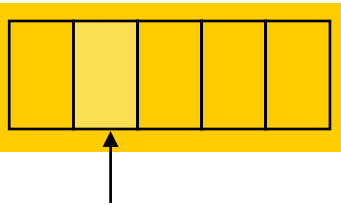


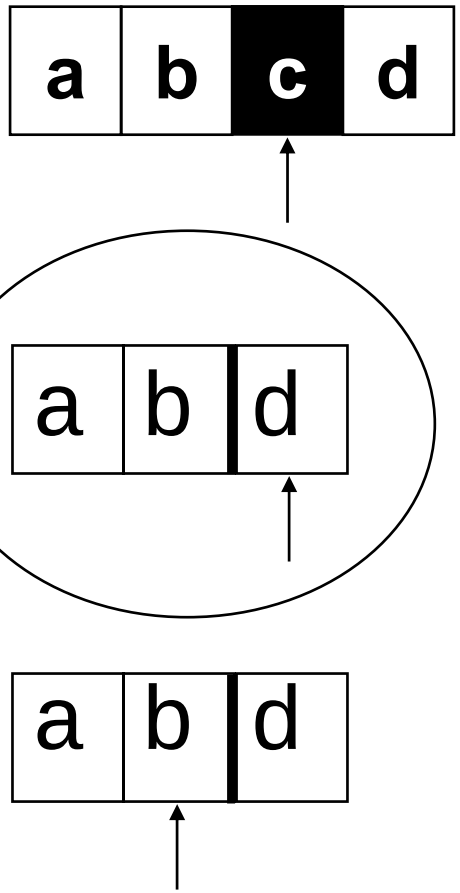
Na vložený prvek?

Nebo nechat na aktuálním?

Vkládáme před aktuální prvek, ukazovátko neměníme

List – delete





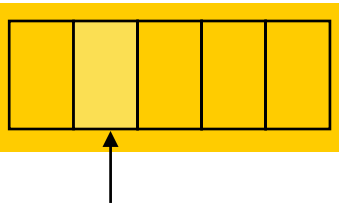
Kam nastavit ukazovátko po smazání?

Na další (*next*)

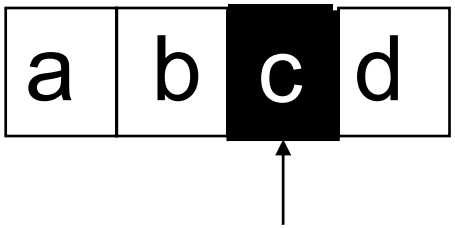
Na předchiozí (*previous*)

Po smazání aktuálního prvku,
ukazovátko posuneme na jeho následníka

List (Seznam)



Příklad: Dva způsoby vytvoření téhož seznamu:

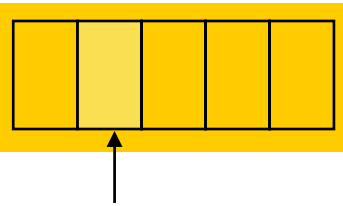



```
insert( b, insert( a,
   first( insert( d, insert( c, init ))
))
```

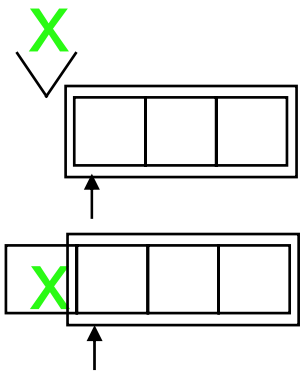
nebo

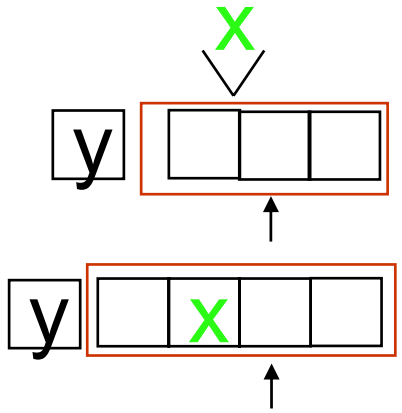
```
prev( prev( insert( d, insert( c,
   insert( b, insert( a, init ))
))))  ... apod…
```

List (Seznam)

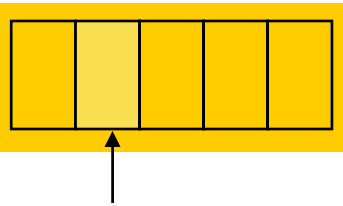


```
var: x,y:Elem, s:Seq, l:List
 init = mark( new )

 insert( x, mark(s) )
  = consL( x, mark(s))
     ... Insert BEFORE the pointer

 insert( x, consL( y, l ))=
  consL( y, insert( x, l ))
```



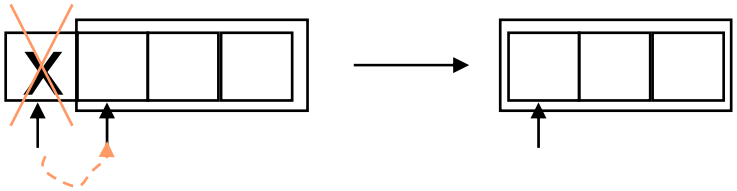


List (Seznam)



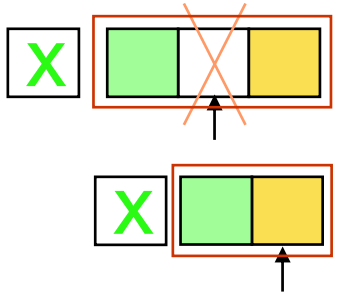
```
delete( init ) = init

delete( mark( cons( x, s)))
   = mark( s )
```

... delete by the pointer

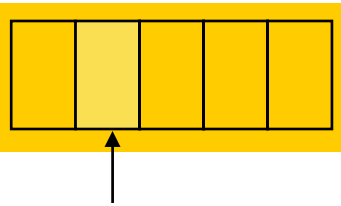



```
delete( consL( x, l )) =
   consL( x, delete( l ))
```

... delete by the pointer

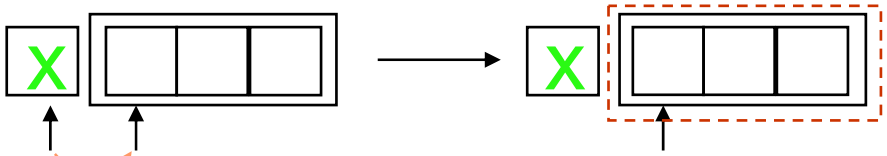


List (Seznam)



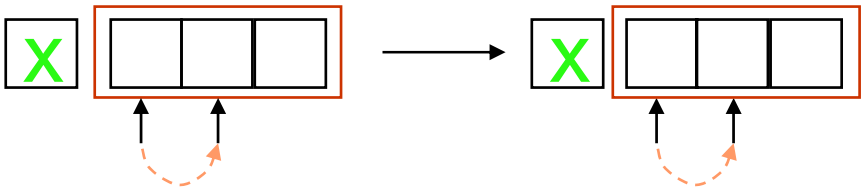
```
next( init ) = init

next( mark( cons( x, s ) ) ) =
   consL( x, mark( s ) )
```

... pointer move

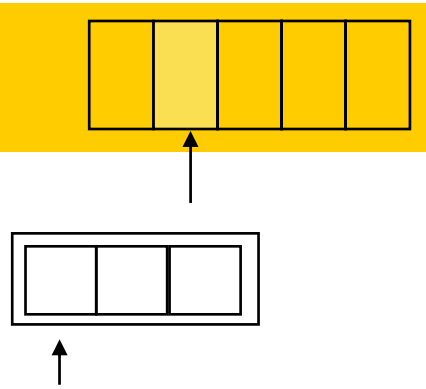



```
next( consL( x, l ) ) =
   consL( x, next( l ) )
```

..no change before



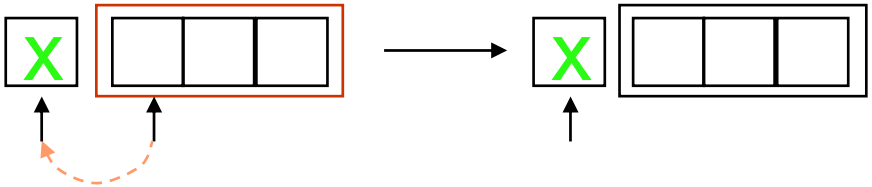
List (Seznam)



```
prev( mark(s) ) = mark(s)
```

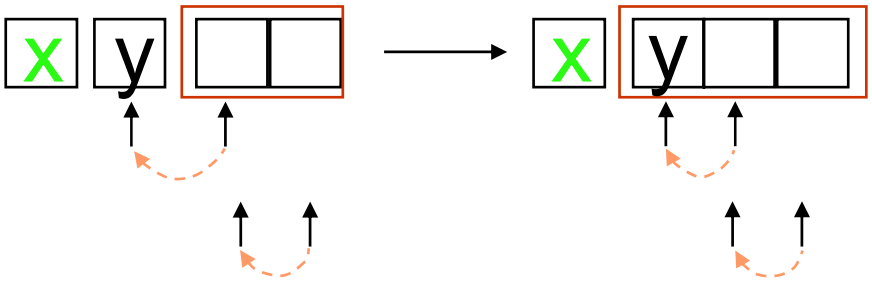
```
prev( consL( x, mark( s ) ) ) =
 mark( cons( x, s ))
```

... move by head

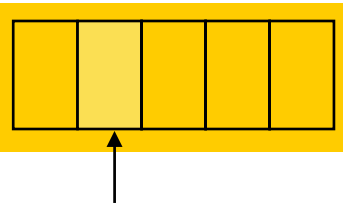



```
prev( consL( x, consL( y, l ))) =
   consL( x, prev( consL( y, l )))
```

... move inside

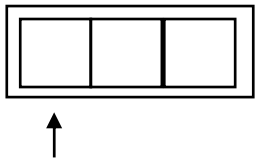


List (Seznam)



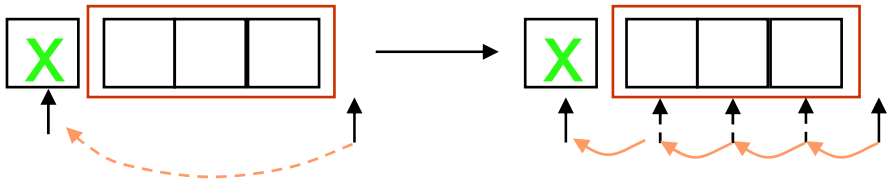
```
first( mark(s) ) = mark(s)
```

... no move by head

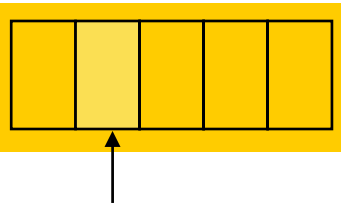


```
first( consL( x, l ) ) =
  first( prev( consL( x, l ) ) )
```

... move step by step to front

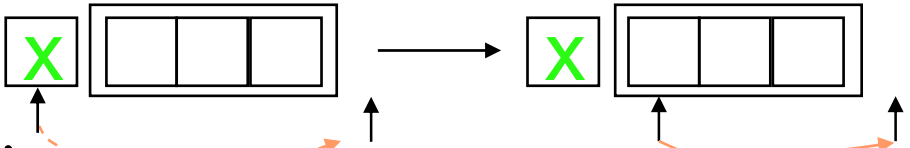


List (Seznam)



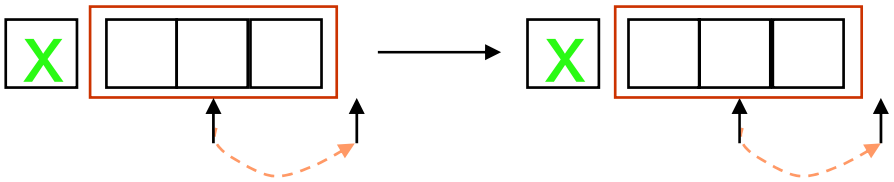
```
last( init ) = init

last( mark( cons( x, s ) ) ) =
   consL( x, last( mark( s ) ) )
```

... move back

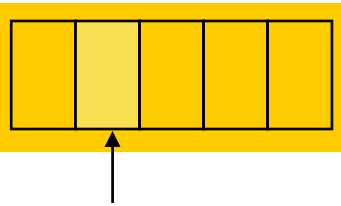



```
last( consL( x, l ) ) =
   consL( x, last( l ) )
```

...no change before

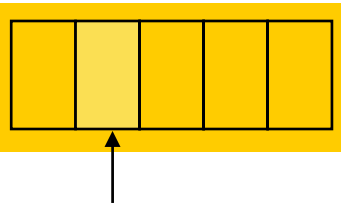


List (Seznam)



```
read( init ) = error_elem

read( consL( x, l )) = read( l )

read( mark( cons( x, s ))) = x
```

List (Seznam)



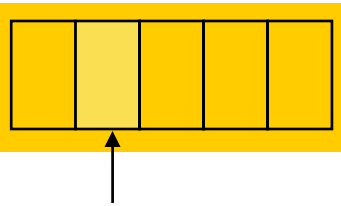
```
length( init ) = 0

length( consL( x, l )) =
   succ( length( l ) )
```

... elements before the pointer

```
length( mark( cons( x, s ))) =
   succ( length( mark( s )))
```

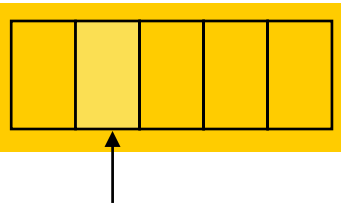
... elements after the pointer

List (Seznam)



```
empty( l ) = ( length( l ) == 0 )

full( l ) = ( length( l ) == max )

atbeg( mark( s )) = true
atbeg( consL( x, l )) = false

atend( mark( new)) = true
atend( mark( cons( x, s )))= false
atend( consL( x, l )) = atend( l )
```

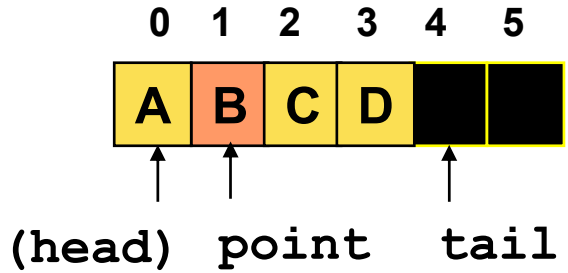
List implementation



In array

O(n) insert, delete

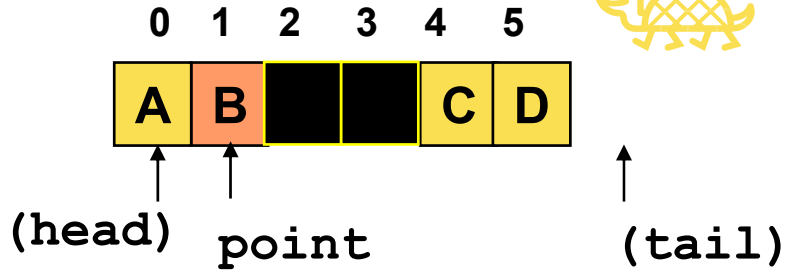
O(1) first, last, prev, next

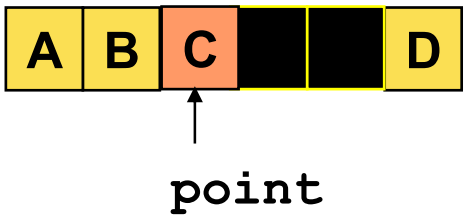


Two Stacks in Array

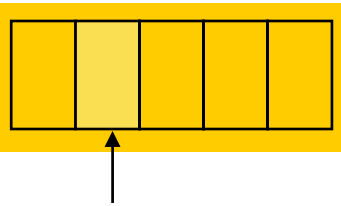
O(1) insert, delete, prev, next

O(n) first, last, …

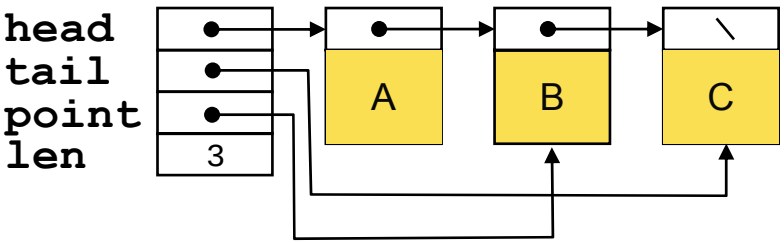


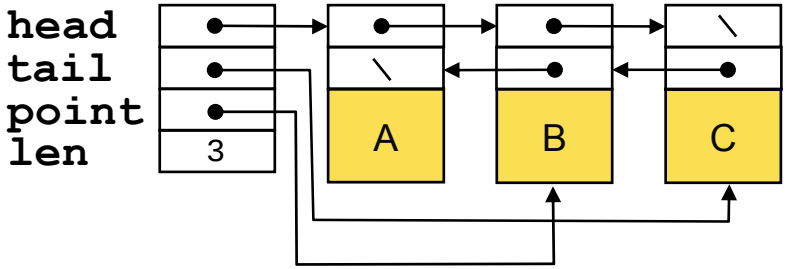


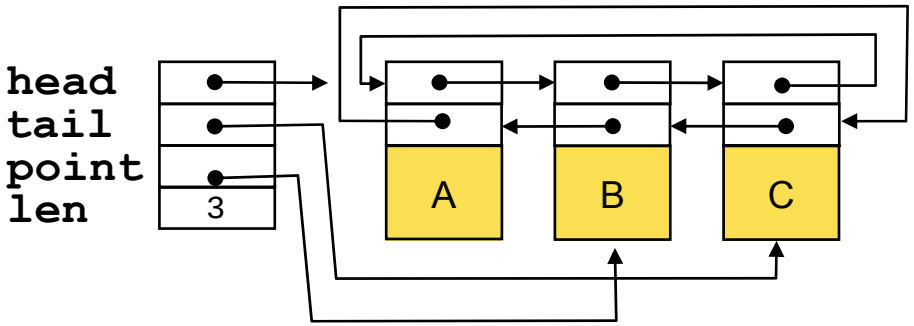
List implementation



Linked In dynamic memory







O(1) insert,

O(1)..O(n) delete

O(1) first, last, prev, next

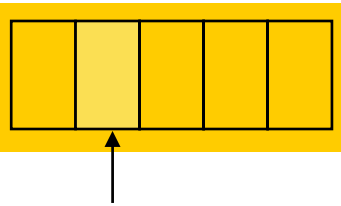
- memory for pointers

Linked list

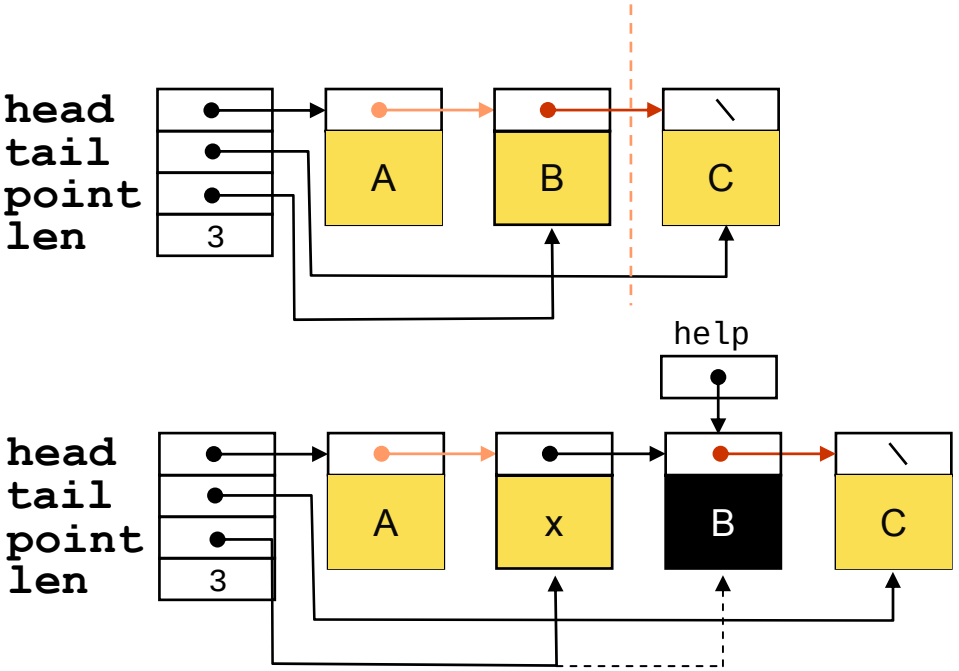
Double linked list

Circular double linked list

List implementation



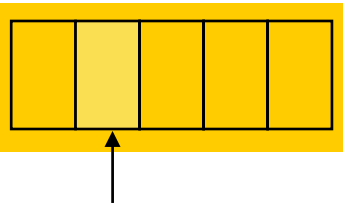
Linked list



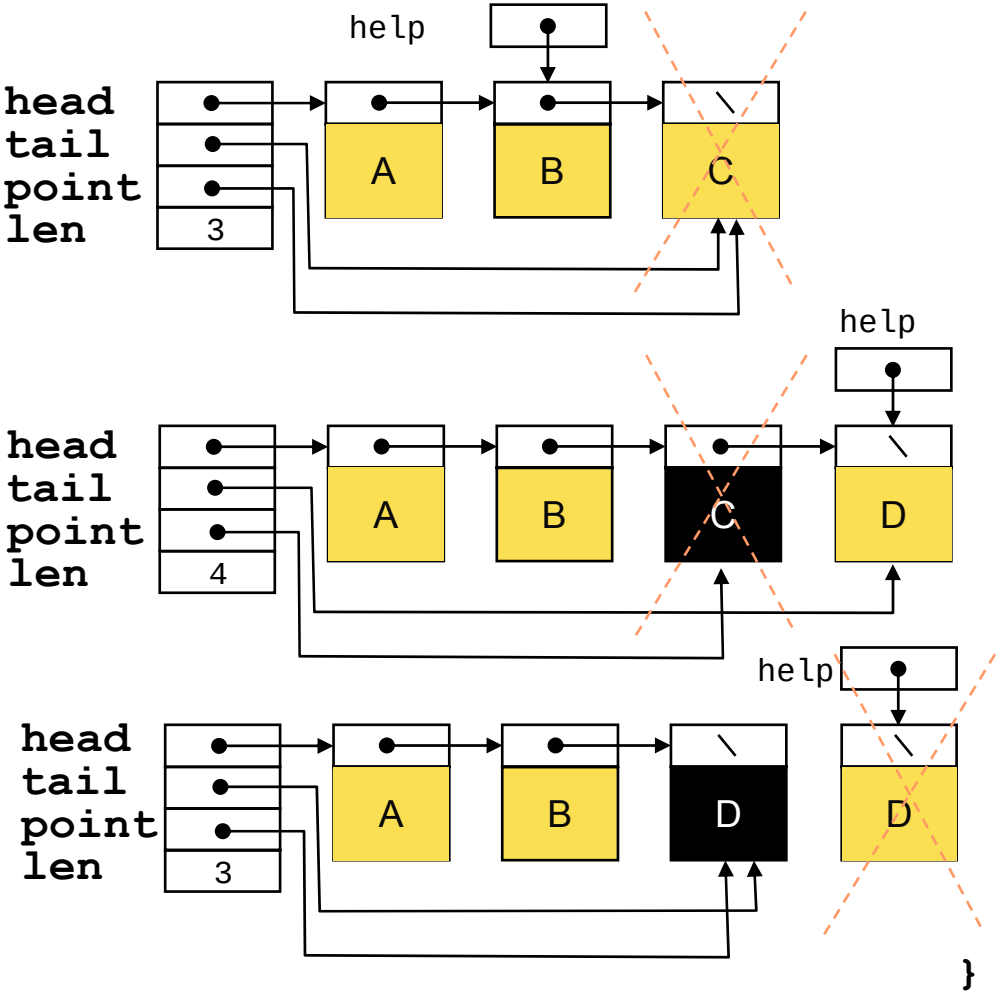
`tail` = **poslední element**
`point` **== NULL ukazuje za** `last`

```
list::insert( elem x ) {

  item *help = new item();

  if( point == NULL ){//point behind
    help->next = NULL;
    help->val = x;
    if( tail == NULL ) //empty list
       head = help;
    else                    //add at end
       tail->next = help;
    tail = help;
  } //point points behind list!

  else { //point in the list - trick
    help->val = point->val;
    help->next = point->next;
    point->next = help;
    point->val = x;
    point = help;
  }
  len++;
}
```

List implementation

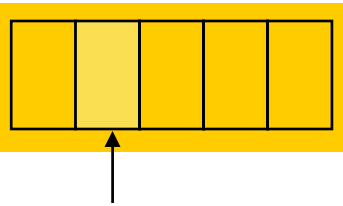


Linked list

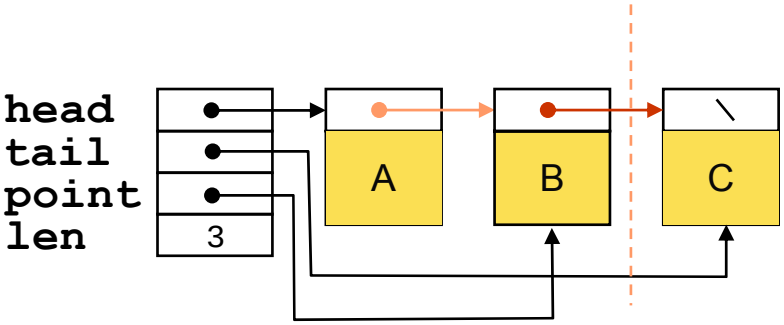



```
list::delete( ) {
  item *help;
  if( point != NULL ){//behind ignore

    if( point->next == NULL ) {//last
      help = head; //find predecessor
      while( help->next != point )
        help = help->next;
      }
      help->next = NULL;
      point = NULL;
      delete( tail ); tail = help;
    }
    // not last
    else {// trick:skip predec.search
      help = point->next;
      *point = *help;
      if( help == tail );
        tail = point;
      delete( help );
    }
    len--;
  }
}
```

List implementation

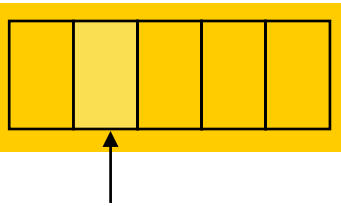


Linked list

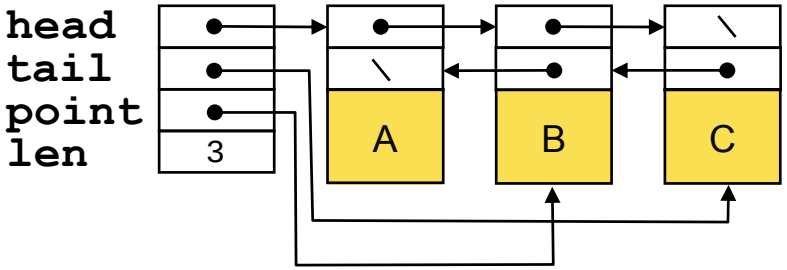



```
list::prev( ) {

  item *help;
  if( point != head){ // could move
    help = head;
    while( help->next != point )
      help = help->next;
    point = help;
  }
}
```

List implementation



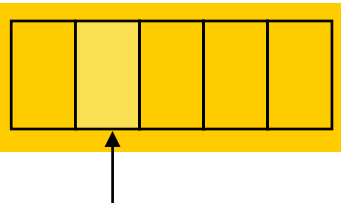
Double linked list



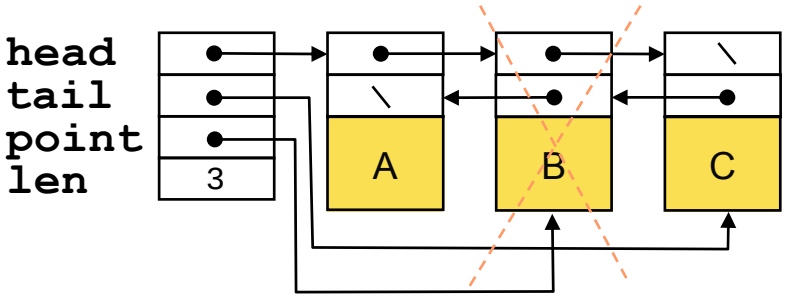

```
list::prev( ) {

  item *help;
  if( point != head){ //could move
    if(point == null)
      point = tail;   // last
    else
      point = point->prev;
  }
}
```

`Prev()` is the only place to save some operations (mininize the complexity)!

List implementation

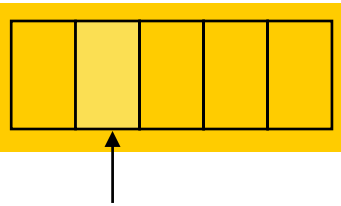


Double linked list

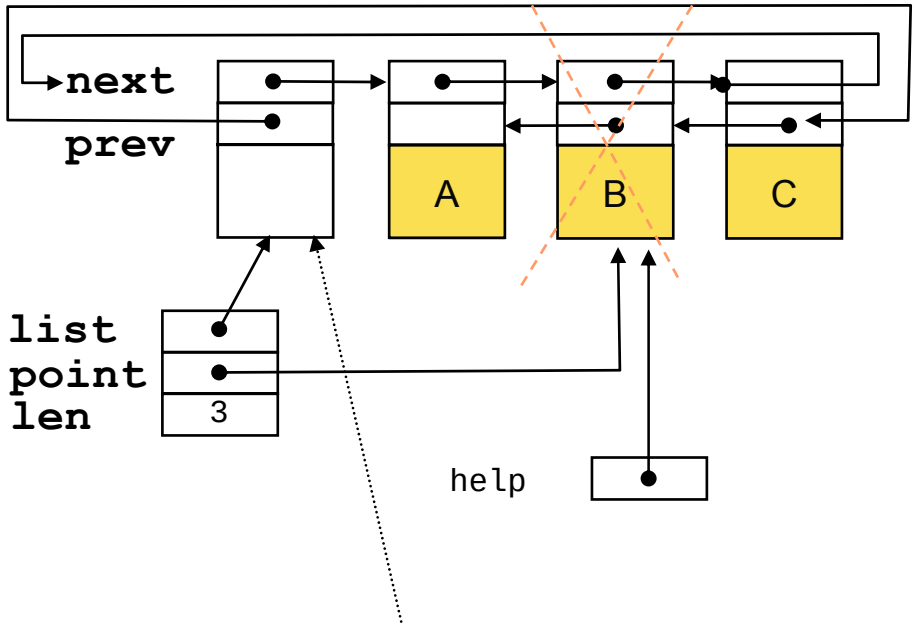



```
list::delete( ) {
  item *help;
  if( point != NULL ){//behind ignore

    help = point->next ;

    if( head == point ) //first
       head = help;

    if( tail == point ) //last
       tail = tail->prev;

    if( help != NULL )  //prev
       help->prev = point->prev;

    if( point->prev != NULL ); //next
       point->prev->next = help;

    delete( point );
    point = help;

    len--;
  }
}
```

List implementation



Circular double linked list



additional item

not used for data

SIMPLIFIES delete()

```
list::delete( ) {
  item *help;
  if( point != list ){ //not at end
    point->prev->next = point->next;
    point->next->prev = point->prev;

    help = point;
    point = point->next;

    delete( help );

    len--;
  }
}

list::prev( ) {
  if( !atBegin() )//point!=list->next
    point = point->prev;
}
```

Abstraktní datové typy

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

Seznam *(List)*

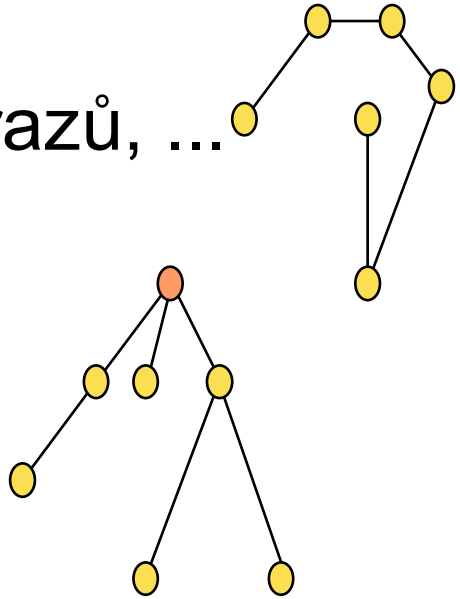
Strom *(Tree)*

Množina *(Set)*

Strom (Tree)

Kdy?

- řazení, vyhledávání, vyhodnocování výrazů, ...
- acyklický souvislý graf
- *kořenový strom*
  - orientovaný strom, zvláštní uzel - *kořen*
  - *kořen* spojen se všemi uzly orient. cestou
  - *binární strom* - má 0, (1), 2 následníky
  - uspořádaný binární strom - dvojice $<u_1, u>$ a $<u, u_2>$ jsou uspořádáné



Strom (Tree)

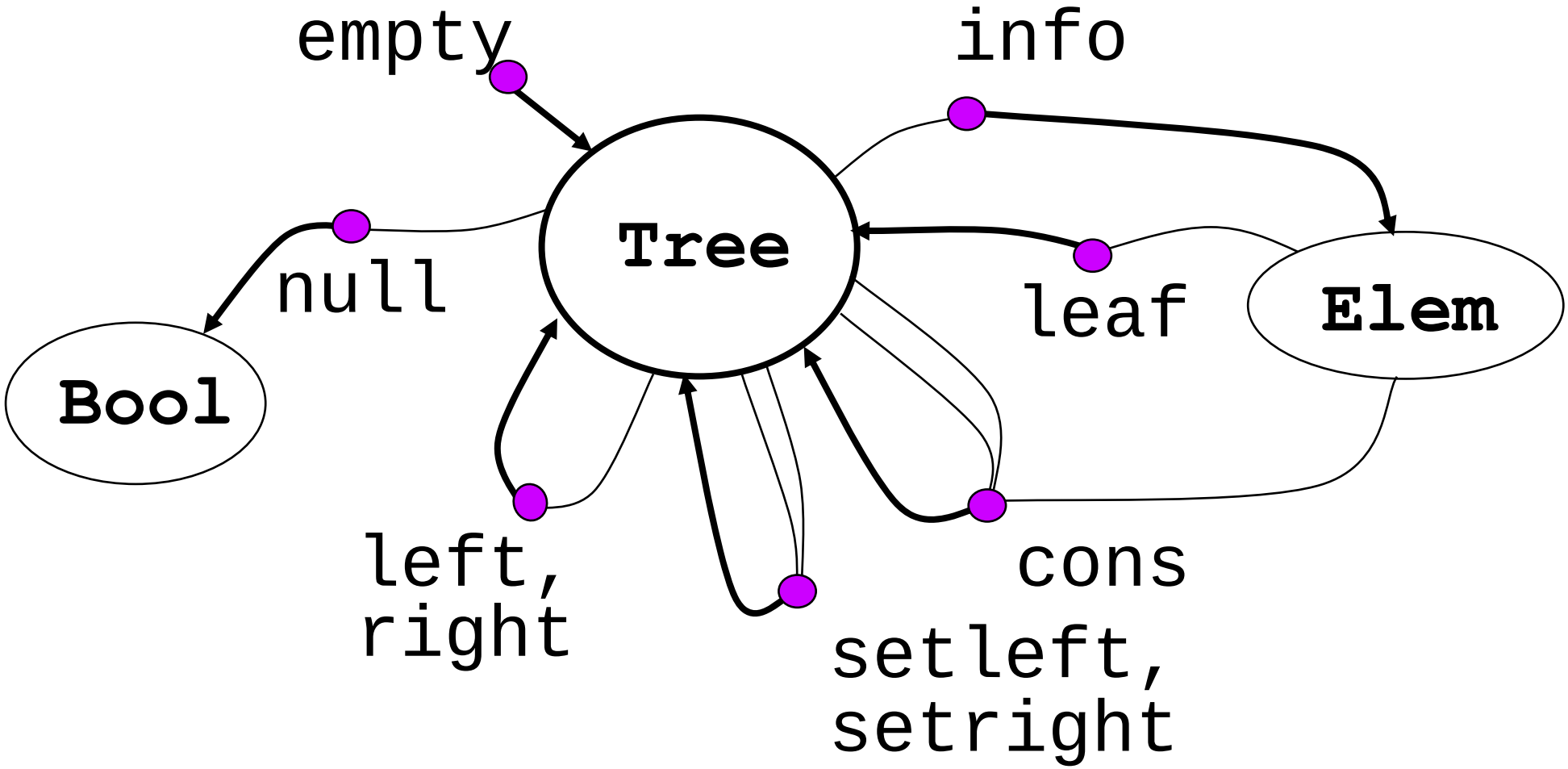
uzel - více následníků

binární strom - 2 následníci

následník - opět strom

homogenní, nelineární, dynamická

Binární strom (Tree)



Binární strom (Tree)

```
empty: -> Tree
leaf(_): Elem -> Tree
cons(_,_,_): Elem, Tree, Tree
                    ->Tree
left(_), right(_): Tree -> Tree
null(_): Tree -> Bool
setleft, setright(_,_): Tree,
  Tree -> Tree
setinfo(_,_): Tree, Elem -> Tree
info(_): Tree -> Elem
```

Binární strom (Tree)

```
var x: Elem; a,b,t: tree

leaf( x ) = cons( x, empty, empty)

left( empty ) = error_tree
left( cons( x, a, b)) = a

right( empty ) = error_tree
right( cons( x, a, b)) = b
```

Binární strom (Tree)

```
null( empty ) = true
null( cons( x, a, b)) = false

setleft( empty, t ) = error_tree
setleft( cons( x, a, b ), t ) =
  cons( x, t, b )

setright( empty, t ) = error_tree
setright( cons( x, a, b ), t ) =
  cons( x, a, t )
```

Binární strom (Tree)

```
setinfo( empty, x ) = error_tree
setinfo( cons( x, a, b ), y ) =
  cons( y, a, b )

info( empty ) = error_elem
info( cons( x, a, b ) ) = x
```

Abstraktní datové typy

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

Seznam *(List)*

Strom *(Tree)*

Množina *(Set)*

Parametrické datové typy

Nové typy

- obohacováním původních o druhy a operace
- často stejné, jen nad jinými základy
- liší se jen některými prvky

=> parametrický datový typ (jako šablona v C++)

odlišné prvky - parametry

Množina

```
typ: MNOŽINA( ELEMENT )
parametr: typ prvků ELEMENT
požadavky na parametr:
      druhy: Elem
      operace: eq(_,_): Elem, Elem -> Bool
použité typy: ELEMENT, přirozená čísla,
       logické hodnoty
druhy: Elem, Bool, Set, Nat
```

Množina

operace:

`[]: Set`		**(prázdná množina)**
`ins(_,_):`	`Elem,Set ->Set`	**(vložení prvku)**
`del(_,_):`	`Elem,Set ->Set`	**(zrušení -"-)**
`in(_,_) :`	`Elem,Set ->Bool`	**(test přísluš.)**
`card(_) :`	`Set->Nat`	**(počet prvků)**
`eql(_,_):`	`Set, Set ->Bool`	**(rovnost mn.)**

Množina bez opakování

- Asociativní Kontejner
- Žádné dva objekty se neopakují

Množina bez opakování

```
proměnné:
      s: Set, x,y: Elem
axiomy:
      ins(x,s) = if in(x,s) then s

      del(x,[])=[]
      del(x,ins(y,s))= if eq(x,y)
                       then del(x,s)
                       else ins(y,del(x,s))
```

bez opakování prvků

nezáleží na pořadí vkládání

Množina bez opakování

```
axiomy: (pokračování)
     in(x,[]) = false
     in(x,ins(y,s))= if eq(x,y)then true
                     else in(x,s)
     card([]) = 0
     card(ins(x,s)) = if(in(x,s)) card(s)
                      else succ(card(s))
```

Množina bez opakování

```
axiomy: (pokračování)
  eql([],[]) = true
  eql([], ins(x, t)) = false
  eql(s,t) = eq(t,s)

  eql(ins(x,s),t)=
      if in(x,s)
     then eql(s,t)
     else if in(x,t)
          then eql(s,del(x,t))
          else false
```

Abstraktní datové typy

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

Seznam *(List)*

Strom *(Tree)*

Množina *(Set)*

Množina s opakováním (MultiSet) *- rozmyslete sami*

Příloha 1 – Výrazy nad signaturou

a) konstanta: f

(f je nulární operace deklarovaná jako f:d)

b) proměnná: x

(x is proměnná deklarovaná jako var x:d)

c) operace v prefixové notaci: $f(t_1,t_2, \ldots t_n)$

(f is n-ární operace deklarovaná jako:

$f: d_1, d_2, \ldots, d_n \rightarrow d$

a $t_1, t_2, \ldots, t_n$ jsou výrazy druhů

$d_1, d_2, \ldots, d_n$)

d) Operace v infixové notaci: $t_1\ f_1\ t_2\ f_2\ \ldots f_n\ t_{n+1}$

(f is (n+1)-ární operace deklarovaná jako:

$\_f_1\ \_f_2\ \_\ldots\_f_n\ \_\_ : d_1, d_2, \ldots, d_{n+1} \rightarrow d$

a $t_1, t_2, \ldots, t_{n+1}$ jsou výrazy druhů $d_1, d_2, \ldots, d_{n+1}$)

e) závorky: (t)

(t je výraz druhu d)

f) Žádný jiný zápis výrazem není

Výrazy (pokrač.):

Herbrandovo universum pro danou signaturu

= množina všech výrazů nad danou signaturou

Příklad:

```
{true, false, not(true), not(false),
not(not(true)),...}
```

Dvě třídy ekvivalence (popisují stejnou hodnotu)

- `{true, not(false), not(not(true)),...}`
  `... [true]`
- `{false, not(true), not(not(false)),...}`
  `... [false]`

Příloha 2 – třídy ekvivalence

Třídy ekvivalence

– Obsahují výrazy, které popisují různý způsob konstrukce stejné hodnoty

– Bool: lze rozdělit na 2 třídy ekvivalence :

- {true, not(false), not(not(true)),...} ... [true]
- {false, not(true), not(not(false)),...}... [false]

Bool - třídy ekvivalence

Formálně bychom měli doplnít axiomy

```
true = [true]
false = [false]
not([x])=[not(x)]
and([x],[y]) = [and(x, y)]
```

Axiomy s třídami ekvivalence
=> tvoří model datového typu

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. (Pozor, má jinak šipky a jiný seznam) 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

Abstraktní datové typy