

# Abstraktní datové typy II.

Karel Richta a kol.

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

© Karel Richta a kol., 2023

Datové struktury a algoritmy, B6B36DSA  
09/2023, Lekce 9

<https://cw.fel.cvut.cz/wiki/courses/b6b36dsa/start>



# Abstraktní datové typy

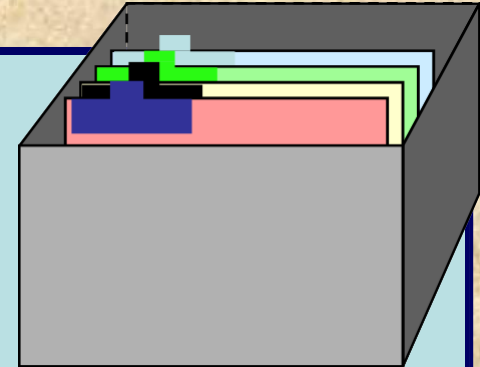
- Pole (*Array*)
- Zásobník (*Stack*)
- Fronta (*Queue*)
- ✓ **Tabulka (*Table*)**
- Seznam (*List*)
- Množina bez opakování (*Set*)
- Množina s opakováním (*MultiSet*)

# Vyhledávací Tabulka (Look-up Table)

- kartotéka, asociativní paměť
- převod mezi kódy, četnost slov,...
- *homogenní, dynamická* (nejen) a nelineární
- obsahuje
  - položky tvořené klíčem a (nřínadnou) asociovanou hodnotou
  - *klíč* jednoznačně identifikuje položku, podle něj se vyhledává

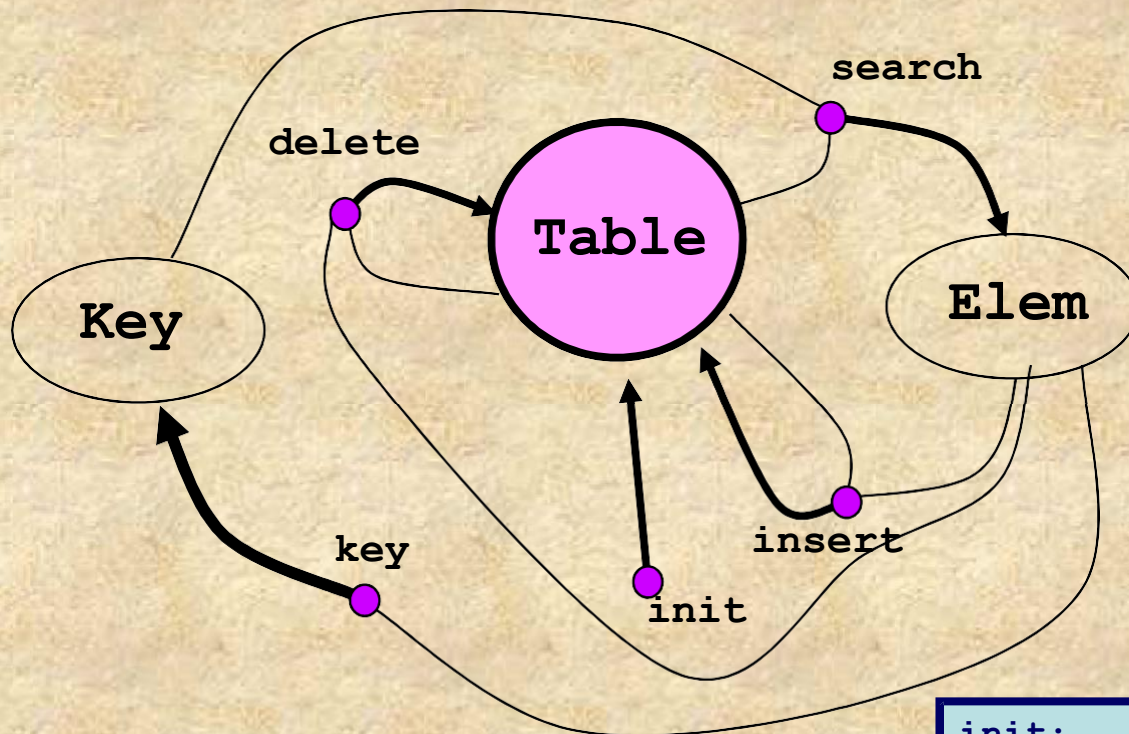
## Příklady:

- **telefonní seznam** - klíčem je jméno+adresa, hodnotou tlf číslo
- **Č-A slovník** - položka je dvojice české slovo+anglické slovo





# Signatura tabulky



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

# Možnosti vyhledávání

## Asociativní vyhledávání - porovnáváním klíčů

- nalezeno, když *klíč\_prvku = hledaný klíč*
- např. sekvenční vyhledávání, hledání půlením, BVS,...

$\Omega(\log n)$

## Adresní vyhledávání

- **indexací klíčem (přímý přístup)**
  - klíč je přímo indexem (adresou)
  - rozsah klíčů ~ rozsahu indexů
- **rozptylováním (hashing)**
  - výpočtem adresy z hodnoty klíče

$\Theta(1)$

průměrně  $\Theta(1)$



# Rozptylování - Hashing

**Rozptylování je kompromis** mezi rychlostí a spotřebou paměti

- sekvenční vyhledávání - čas  $O(n)$ , paměť  $O(n)$
- přímý přístup - čas  $O(1)$ , paměť  $O(N)$ ,  $N = |\text{universum klíčů}|$
- rozptylování stačí málo času i paměti
  - velikost tabulky  $m$  reguluje čas vyhledání
  - pro  $m = kn$  je čas  $O(1)$
- **konstantní očekávaný čas** pro vyhledání a vkládání (*search a insert*) !

**Jaká je cena? (trade-off)**

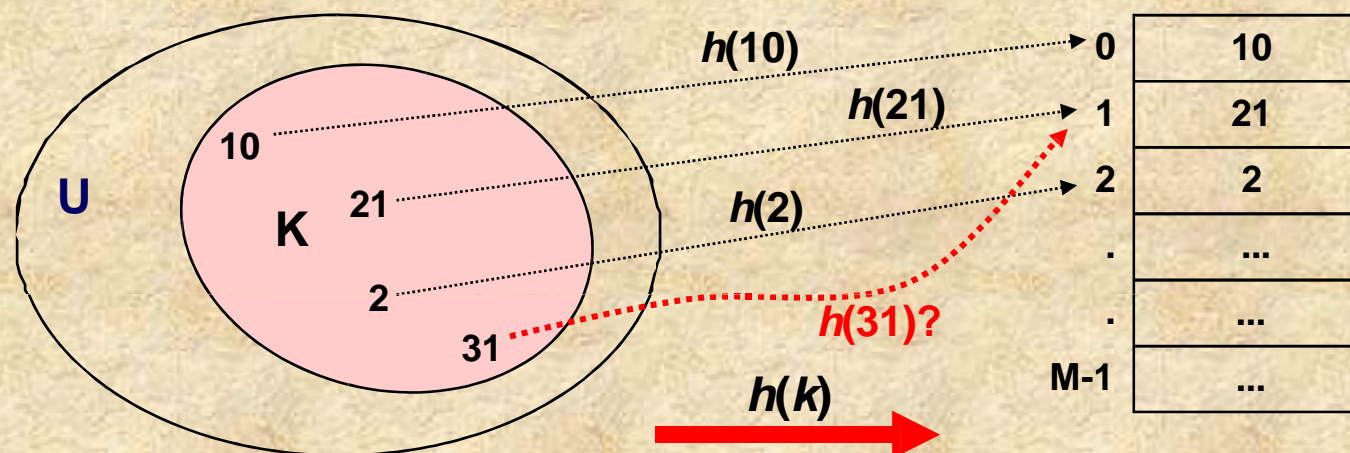
- čas provádění  $\sim$  délce klíče
- nevhodné pro operace *výběru  $k$ -tého prvku (select) a řazení (sort)*

# Rozptylování

U - universum klíčů

K – skutečně použité klíče

$|K| \ll |U|$



Rozptylování má **dvě fáze**

1. Výpočet **rozptylovací funkce**  $h(k)$  (z hodnoty klíče vypočítá adresu)
2. Vyřešení **kolizí** (  $h(31) = h(21)$  ... **kolize** – prvek s indexem 1 již obsazen)

# Rozptylovací funkce $h(k)$

## Rozptylovací funkce

- zobrazuje množinu klíčů  $K \subseteq U$  do intervalu adres  $A = \langle \min, \max \rangle$ , obvykle  $\langle 0, M-1 \rangle$
- $|U| \gg |K| \approx |A|$
- vznikají tak nutně **synonyma**:  $h(k_1) = h(k_2)$  pro  $k_1 \neq k_2$ ,  
 $\Rightarrow$  nastane **kolize** (místo v tabulce obsazeno)  
**Př.:** Pro  $h(k) = k \bmod 5$  jsou synonyma (10, 20, 55, ...) (2, 12, 17, 22, ...)
- silně závisí na vlastnostech klíčů a jejich reprezentaci v paměti
- ideálně:
  - výpočetně co nejjednodušší (rychlá)
  - aproximuje náhodnou funkci
  - využije **rovnoměrně** adresní prostor
  - generuje **minimum kolizí**
  - proto: měla by využívat **všechny části klíče**



# Volba rozptylovací funkce $h(k)$

- Pro **reálná čísla** z intervalu  $\langle 0,1 \rangle$  (jak se na něj převede interval  $\langle a,b \rangle$  ?)
  - multiplikativní:  $h(k,M) = \text{round}(k * M)$  (neoddělí shluky blízkých čísel)  
 $M$  = velikost tabulky
- Pro  $w$ -bitová **celá čísla**
  - multiplikativní: ( $M$  je prvočíslo)
    - $h(k,M) = \text{round}(k / 2^w * M)$
  - modulární:
    - $h(k,M) = k \% M$  (**pozor na volbu modulu  $M$ !!**)
  - kombinovaná:
    - $h(k,M) = \text{round}(c * k) \% M, c \in \langle 0,1 \rangle$
    - $h(k,M) = (\text{int})(0.616161 * (\text{float}) k) \% M$
    - $h(k,M) = (16161 * (\text{unsigned}) k) \% M$
- Pro **obecné hodnoty** (rychlá, silně závislá na reprezentaci klíčů)
  - $M$  bitů z klíče
    - $h(k,M) = k \& (M-1)$  pro  $M = 2^x$  (není prvočíslo!),  $\&$  je bitový součin

# Volba rozptylovací funkce $h(k)$

- Pro řetězce  $c_k c_{k-1} \dots c_2 c_1 c_0$  – bereme jako polynom  $\Rightarrow$  použije se **Hornerovo schéma**

$$c_k * a^k + c_{k-1} * a^{k-1} + \dots + c_2 * a^2 + c_1 * a^1 + c_0 * a^0 = (((((c_k * a + c_{k-1}) * a \dots + c_2) * a + c_1) * a + c_0$$

```
static int hash( String s, int M) {  
    int h = 0, a = 127;   
    for (int i = 0; i < s.length(); i++)  
        h = (a*h + s.charAt(i)) % M;  
    return h;  
}
```

**a=127 je vhodný základ  
(např. 128 není vhodné)**

- Pro řetězce (pseudo-)randomizovaná tzv. **univerzální rozptylovací funkce**

```
static int hashU( String s, int M) {  
    int h = 0, a = 31415, b = 27183;  
    for (int i = 0; i < s.length(); i++) {  
        h = (a*h + s.charAt(i)) % M;  
        a = a*b % (M-1);  
    }  
    return h;  
}
```

// pseudonáhodná posloupnost

# Řešení kolizí

**Jak postupovat, když dojde ke kolizi?**

Dva přístupy:

- **řetězení**
- **otevřené rozptylování / adresování**

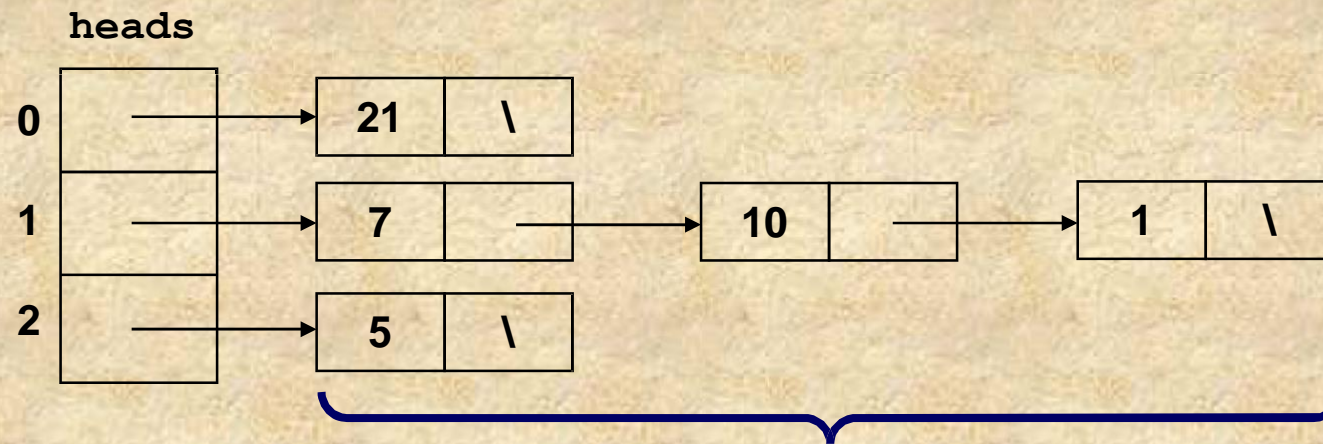


# Řešení kolizí řetězením (Chaining)

Předpokládejme:  $h(k) = k \bmod 3$

- vkládáme posloupnost klíčů: 1, 5, 21, 10, 7
- nový klíč vždy na začátek řetězu (proč asi?)

$m = 3, n = 5$



seznamy synonym

# Řešení kolizí řetězením (Chaining)

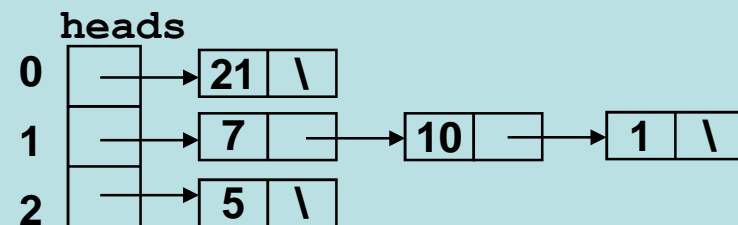
```
private Node[] heads;
int N, M;

void init( int maxN ) {           // initialization
    N=0;                          // # of table items
    M = maxN / 5;                 // table size
    heads = new Node[M];         // array of pointers
    for( int i = 0; i < M; i++ )
        heads[i] = null;
}

Elem search( Key k ) {           // search in the respective sublist
    return seqSearchList( heads[hash(k, M)], k
    );
}

void insert( Elem item ) {
    int i = hash( item.key(), M );
    heads[i] = new Node( item, heads[i]
    ); N++;
}

void delete( Elem item ) {
    ...
}
```

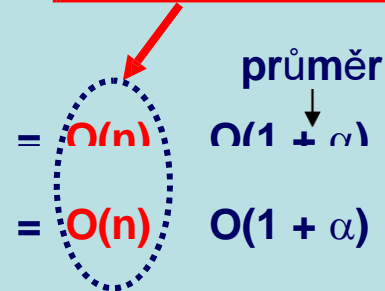


# Řešení kolizí řetězením (Chaining)

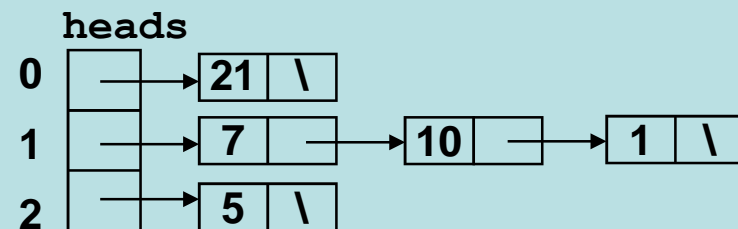
Řetěz synonym má ideálně délku  $\alpha = n / m, \alpha > 1$  (zaplnění tabulky)  
 ( $n$  = počet prvků,  $m$  = velikost tabulky,  $m < n$ )

- **Insert**  $I(n) = t_{\text{hash}} + t_{\text{link}} = O(1)$
- **Search**  $Q(n) = t_{\text{hash}} + t_{\text{search}}$   
 $= t_{\text{hash}} + t_c \cdot n / (2m) = O(n)$   $O(1 + \alpha)$
- **Delete**  $D(n) = t_{\text{hash}} + t_{\text{search}} + t_{\text{link}} = O(n)$   $O(1 + \alpha)$

nepravděpodobný extrém



- pro malá  $\alpha$  (velká  $m$ ) - se hodně blíží  $O(1)$  !!!
- pro velká  $\alpha$  (malá  $m$ ) -  $m$ -násobné zrychlení proti sekvenčnímu hledání





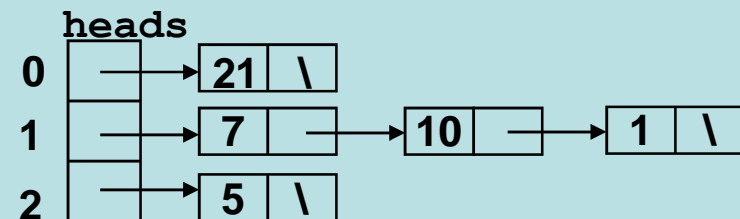
# Řešení kolizí řetězením (Chaining)

## Praxe:

- volit  $m = n/5$  až  $n/10 \Rightarrow$  plnění  $\alpha \leq 10$  prvků / řetěz
- vyplatí se hledat sekvenčně
- neplýtvá nepoužitými ukazateli

## Shrnutí:

- + nemusíme znát  $n$  předem
- potřebuje dynamické přidělování paměti
- potřebuje paměť na ukazatele a na tabulku  $heads[m]$



# Otevřené rozptylování (open addressing)

## Východiska:

- známe (odhadneme) předem počet prvků
- nechceme seznamy ani pole ukazatelů
- $\Rightarrow$  jednotlivé položky tabulky ukládáme (**nesouvisle!**) do pole na místa určená rozptylovací funkcí  $h(k)$

## Jak postupujeme při kolizi?

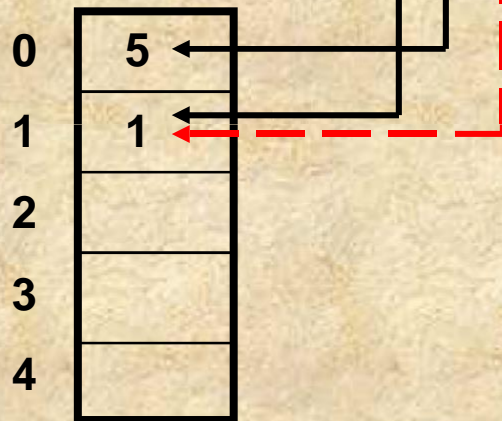
- **lineární prohledávání (linear probing)**
- **dvojití rozptylování (double hashing)**

0	5
1	1
2	21
3	10
4	

# Otevřené rozptylování

## Příklad

- $h(k) = k \bmod 5$  ( $h(k) = k \bmod m$ ,  $m$  je rozměr pole)
- posloupnost: 1, 5, 21, 10



## Problém:

**kolize** – klíč 1 blokuje místo pro klíč 21

## Řešení:

- lineární prohledávání
- dvojí rozptylování

## Poznámka:

1 a 21 jsou synonyma, často ale blokuje nesynonymum.

**Kolize** je blokování **libovolným klíčem**

**Test** (Probe) = určení, zda pozice v tabulce obsahuje klíč shodný s hledaným klíčem

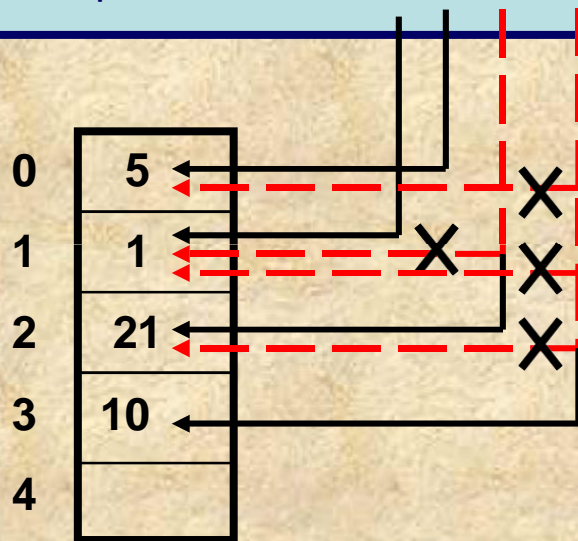
- search hit = klíč nalezen
- search miss = pozice prázdná, klíč nenalezen
- jinak = na pozici je jiný klíč, hledej dál



# Lineární prohledávání

## Příklad

- $h(k,i) = (k+i) \bmod 5$
  - posloupnost: 1, 5, 21, 10, 7
- $(h(k,i) = (k+i) \bmod m, i \text{ je pořadí testu/probe})$



### Problém pro 21:

1 blokuje místo pro 21 – **vlož o pozici dál**

### Problém pro 10:

5 blokuje místo pro 10 – **vlož o pozici dál**

1 blokuje – **vlož o pozici dál**

21 blokuje – **vlož o pozici dál**

# Lineární prohledávání

```
private Elem[] tab;
private int N, M;

void init( int maxN )
{ N = 0; M = 2*maxN; tab = new Elem[M];
}

void insert( Elem x ) {
    int i = hash(x.key(), M);           // co kdyz je tabulka plna ???
    while (tab[i] != null) i = (i+1) %
M; tab[i] = x; N++;
}

int i = hash(key, M);
while (tab[i] != null)                 // co kdyz je tabulka plna ???
    if (key == tab[i].key()) return tab[i];
    else i = (i+1) % M;
return null;
}
```

## Na co je třeba dát pozor:

- lineární prohledávání má **cyklický charakter** (0, 1, 2, ..., M-1, 0, 1, 2, ...)
- musíme se někdy zastavit
  - na volném místě
  - nebo po projití celé (plné) tabulky!
- jak udělat operaci **delete** ?? (viz dále)

# Dvojité rozptylování

Základní myšlenka – obecnější tvar, dvě rozptylovací funkce

$$h(k,i) = ( h_1(k) + i \cdot h_2(k) ) \bmod m$$

**Možná volba**

- $h_1(k) = k \bmod m$  // počátek
- $h_2(k) = 1 + (k \bmod m')$  // offset
- $m =$  prvočíslo  $2^{**}w$  liché
- $m' =$  o něco menší číslo
- nechť  $d = \text{NSD}(m, m') \Rightarrow$  prohledává se jen  $m/d$  položek!



Každý klíč má  
svoji testovací  
posloupnost !

**Příklad:**

- $k = 123456, m = 701, m' = 700, d = \text{NSD}(701, 700) = 1$
- $h_1(k) = 80, h_2(k) = 257$
- hledání začne na prvku 80 a pokračuje s krokem  $257 \% 701$



# Dvojité rozptylování

```
void insert( Elem x ) {
    Key key = x.key();
    int i = hash1(key, M); int k = hash2(key);
    while (tab[i] != null) i = (i+k) % M; // co když je tabulka plná ???
    tab[i] = x; N++;
}

Elem search( Key key ) {
    int i = hash(key, M); int k = hash2(key);
    while (tab[i] != null) // co když je tabulka plná ???
        if (key == tab[i].key()) return tab[i];
        else i = (i+k) % M;
    return null;
}
```

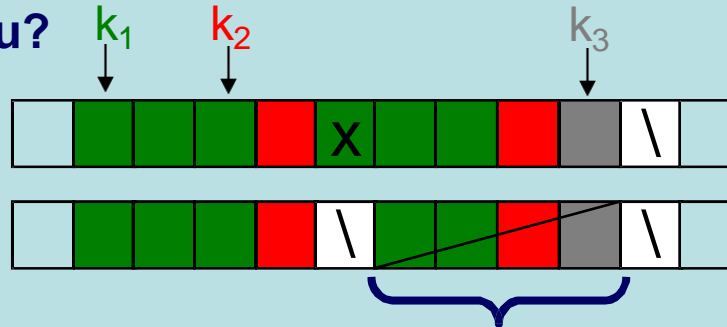
## Na co je třeba dát pozor:

- co dělat při zaplnění tabulky?
  - signalizovat chybu
  - dynamicky zvětšit tabulku
- co to jsou **shluky** (clusters)?
- jak udělat operaci **delete** ?? (viz dále)

# Vypuštění prvku z tabulky (delete)

## Co dělat při vypuštění prvku?

- x nahradíme `null`
- null přeruší shluk(-y) !!!
- => `delete` nesmí nechat "díru"



původní stav

přerušení shluku

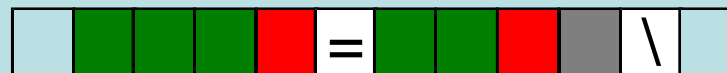
nedosažitelná část shluku

## Řešení se liší:

- pro **lineární** prohledávání  
**znovu vložíme** prvky následující za x (až k prvnímu `null` = konec shluku)



- pro **dvojitě rozptylování**  
vypneme uvolněné místo **speciální zarážkou** .....  
`search` místo přeskočí, `insert` je použije



# Implementace operace delete

```
// delete pro linearni prohledavani
void delete( Elem item ) {
    Key key = item.key();
    int i = hash( key, M );
    while( tab[i] != null )                // find item to remove
        if( item.key() == tab[i].key() ) break;
        else i = (i+1) % M;
    if( tab[i] == null ) return;           // not found
    tab[i] = null; N--;                    //delete, reduce count
    for(int j = (i+1) % M; tab[j] != null; j = (j+1) % M) {
        x = tab[j]; tab[j] = null; insert(x); //reinsert elements after deleted
    }
}

// delete pro dvojite rozptylovani
void delete( Elem item ) {
    Key key = item.key();
    int i = hash1( key, M ), j = hash2( key );
    while( tab[i] != null )                // find item to remove
        if( item.key() == tab[i].key() ) break;
        else i = (i+j) % M;
    if( tab[i] == null ) return;           // not found
    tab[i] = sentinelItem; N--;            // "delete" = replace
}
}
```



# Parametry otevřeného rozptylování

**Průměrný počet testů:**  $\alpha = n / m$ ,  $\alpha \in \langle 0,1 \rangle$  koeficient (za-)plnění

- **Linear probing:**

- Search hits  $0.5 ( 1 + 1 / (1 - \alpha) )$  found
- Search misses  $0.5 ( 1 + 1 / (1 - \alpha)^2 )$  not found

- **Double hashing:**

- Search hits  $(1 / \alpha) \ln ( 1 / (1 - \alpha) ) + (1 / \alpha)$
- Search misses  $1 / (1 - \alpha)$

## Očekávaný počet testů

- **Linear probing:**

Plnění $\alpha$	1/2	2/3	3/4	9/10
Search hit	1.5	2.0	3.0	5.5
Search miss	2.5	5.0	8.5	55.5

- **Double hashing:**

Plnění $\alpha$	1/2	2/3	3/4	9/10
Search hit	1.4	1.6	1.8	2.6
Search miss	1.5	2.0	3.0	5.5

⇒ tabulka může být více zaplněná než začne klesat výkonnost, nebo k dosažení stejného výkonu stačí menší tabulka.

# Reference

- [Cormen] Cormen, Leiserson, Rivest: *Introduction to Algorithms*, Chapter 12, McGraw Hill, 1990
- [Wiki] "Hash function," *Wikipedia, The Free Encyclopedia*,  
[http://en.wikipedia.org/w/index.php?title=Hash\\_function&oldid=175698983](http://en.wikipedia.org/w/index.php?title=Hash_function&oldid=175698983)
- Tables and Hashing presentation  
<http://users.aber.ac.uk/smg/Modules/CO21120-April-2003/NOTES/40-Hashing.ppt>
- Bob Jenkins: Hash Functions for Hash Table Lookup (*theory of hash functions*),  
<http://burtleburtle.net/bob/hash/evahash.html>
- Bob Jenkins: A Hash Function for Hash Table Lookup (*practical examples of hash functions*),  
<http://burtleburtle.net/bob/hash/doobs.html>

# Abstraktní datové typy

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



# Seznam (List)

## Použití

- Sekvenční kontejner, optimalizovaný na vkládání a mazání uvnitř
- Patří mezi nejzákladnější DS ve výpočetní technice (používá se k implementaci jiných DS, stack, queue,...)

## Vlastnosti

- Kontejner s rychlým přístupem ke všem prvkům bez upřednostnění konců
- Optimalizovaný pro vkládání a mazání prvků v libovolné pozici – v místě **ukazovátka**
- Nemá možnost indexovaného přístupu

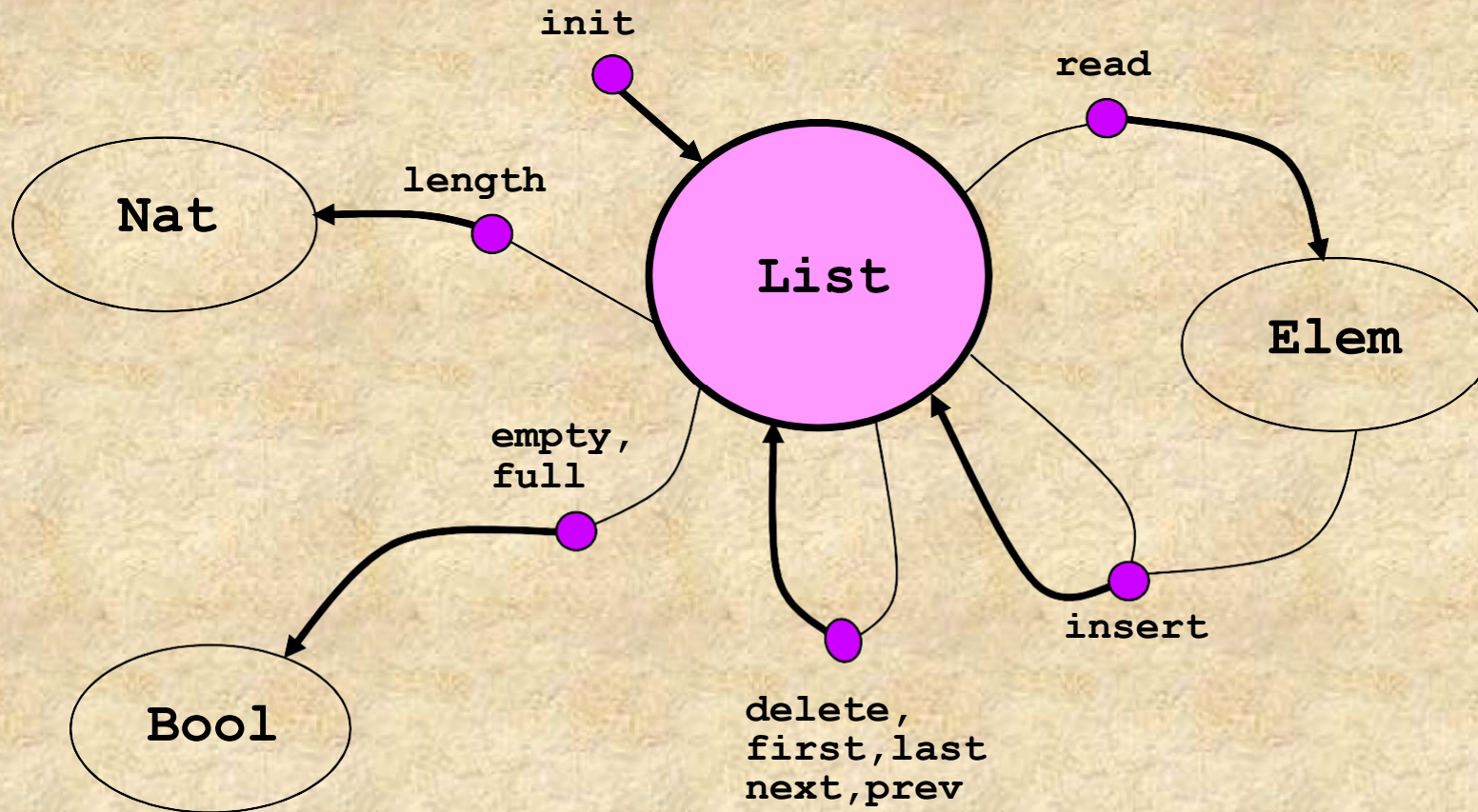
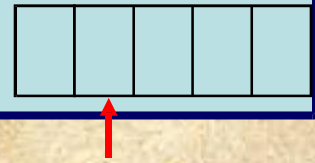
## Intuitivní charakteristika: posloupnost údajů + ukazovátko!

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

Homogenní, lineární, dynamická datová struktura

**Příklad:** spojový seznam = seznam v dynamické paměti (STL)  $O(1)$   
(**NE** ArrayList v Javě, který má  $get(i)$  se složitostí  $O(n)$ )

# Signatura seznamu





# Seznam – interpretace operací



## Operace insert

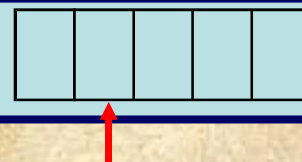
- budeme vkládat **před** nebo **za** ukazovátko?
  - jak se bude vkládat na konec seznamu?
- bude po vložení ukazovátko na **původním** nebo **vloženém** prvku?

## Operace delete

- bude po vymazání ukazovátko na **předchozím** nebo **následujícím** prvku?

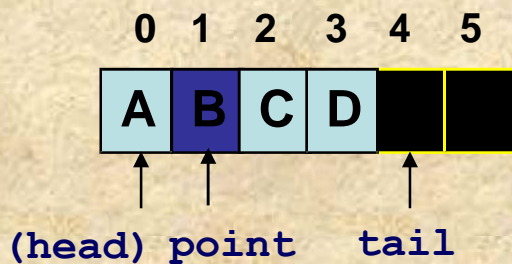


# Implementace seznamu



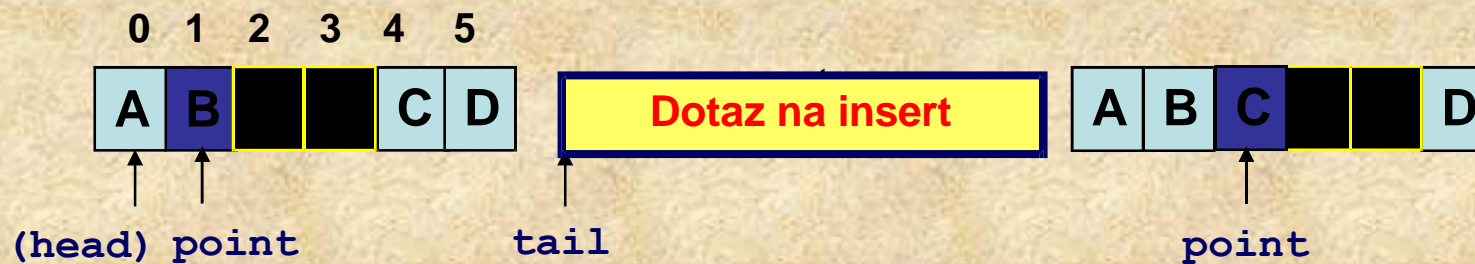
Pomocí pole:

$O(n)$  insert, delete  
 $O(1)$  first, last, prev, next

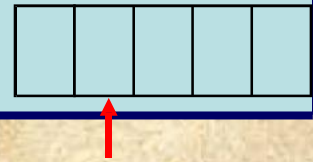


Dva zásobníky v poli:

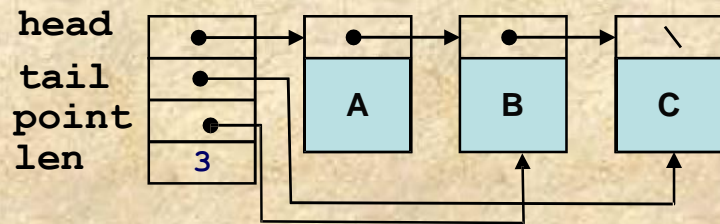
$O(1)$  insert, delete, prev, next  
 $O(n)$  first, last



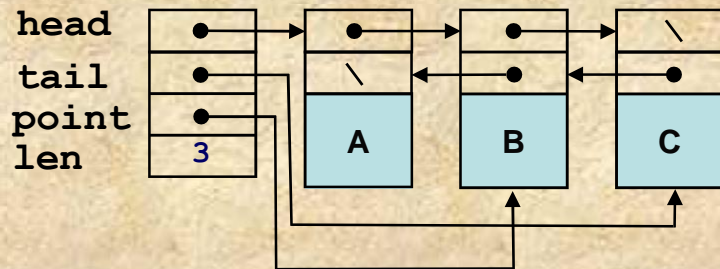
# Implementace seznamu



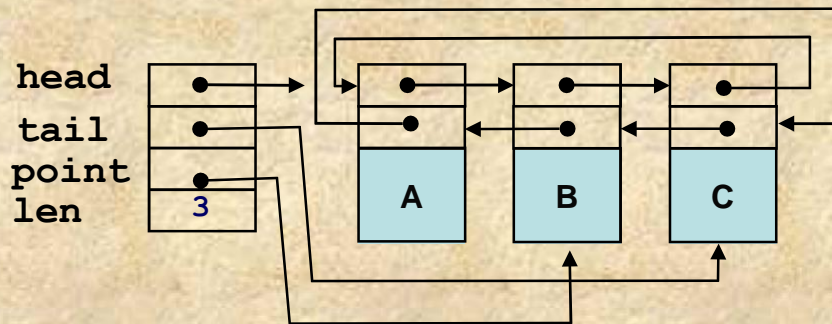
**V dynamické paměti:**  $O(n)$  delete, prev  
 $O(1)$  insert, first, last, next



**Jednosměrně zřetězený seznam**



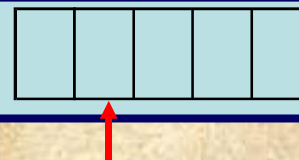
**Obousměrně zřetězený seznam**  
 $O(1)$  delete, prev



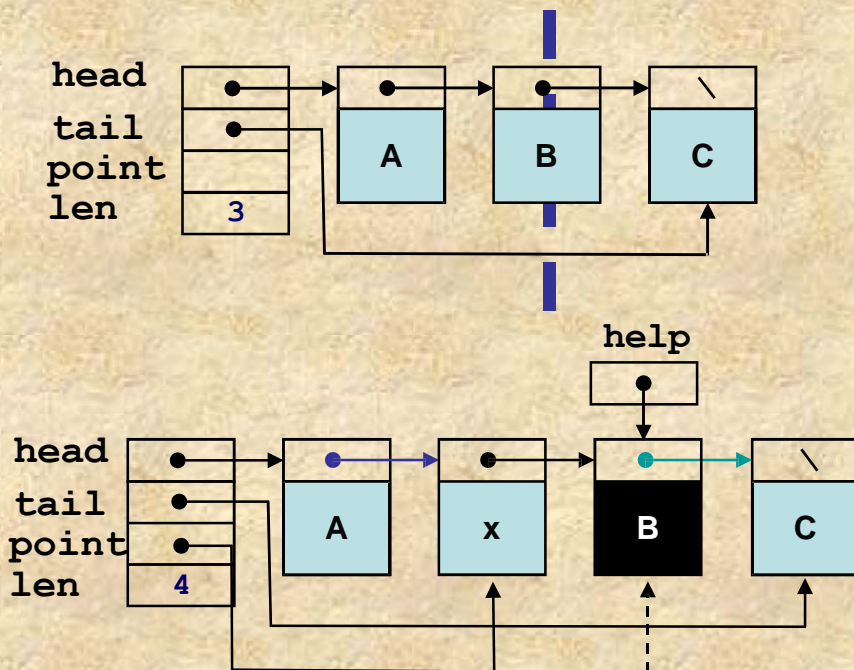
**Kruhově obousměrně zřetězený seznam**



# Implementace seznamu



## Jednosměrně zřetězený



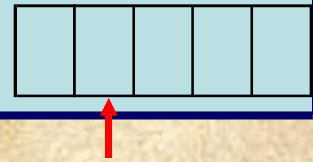
### Konvence

tail = poslední prvek  
point == null **ukazuje** za last

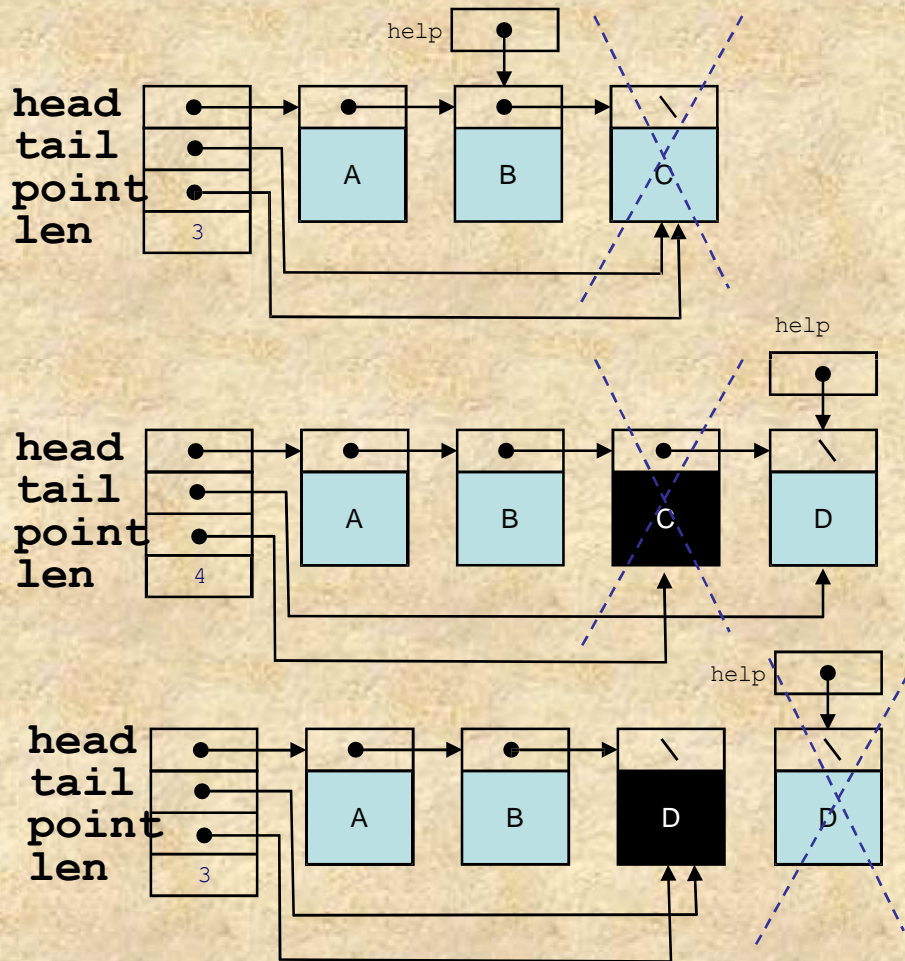
```
void insert( Elem x ) {  
    Node help = new Node();  
    if( point == null ){ // points behind  
        help.next = null;  
        help.val = x;  
        if( tail == null ) // empty list  
            head = help;  
        else // add at end  
            tail.next = help;  
        tail = help;  
    } //point pointed behind list!  
    else { //point is in the list -  
        trick  
        help.val = point.val;  
        help.next = point.next;  
        point.next = help;  
        point.val = x;  
        point = help;  
    }  
    len++;  
}
```



# Implementace seznamu



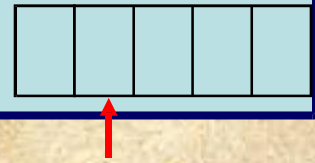
## Jednosměrně zřetězený



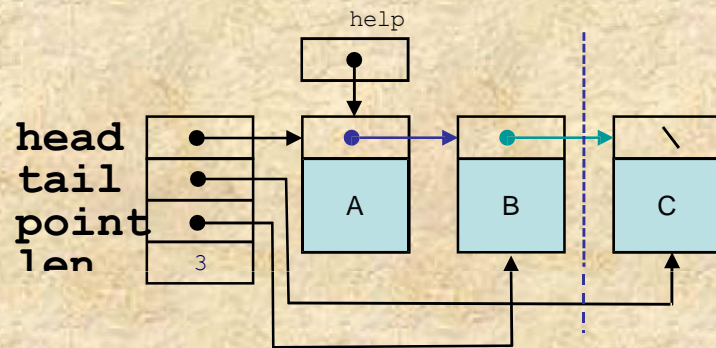
```
void delete( ) {  
    Node help;  
    if( point != null ){ // behind  
        ignored if( point.next == null ) {  
            //help = head; //find predecessor  
            while( help.next != point  
                ) help = help.next;  
        }  
        help.next = null;  
        point = null;  
        tail = help;  
    }  
    // not last  
    else { // trick: skip predec. search  
        help = point.next;  
        point.next = help.next;  
        point.val = help.val;  
        if( help == tail )  
            tail = point;  
    }  
    len--;  
}
```

$O(n)$  mazání tail  
 $O(1)$  mazání uvnitř

# Implementace seznamu



Jednosměrně zřetězený

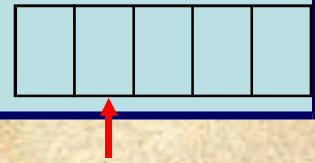


```
void prev( ) {  
    Node help;  
    if( point != head){ // could move  
        help = head;  
        while( help.next != point  
            ) help = help.next;  
        point = help;  
    }  
}
```

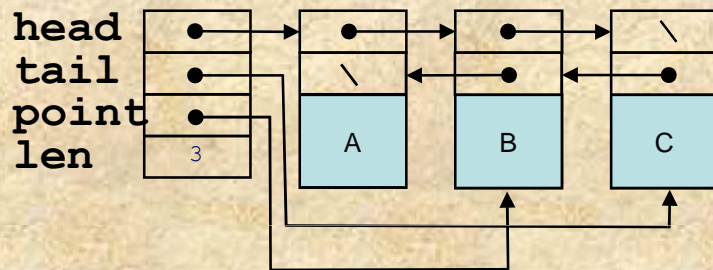
$O(n)$



# Implementace seznamu



## Obousměrně zřetězený



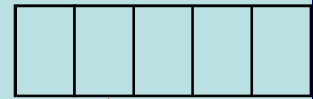
```
void prev( ) {  
    Node help;  
    if( point != head){ //could move  
        if(point == null)  
            point = tail; // last  
        else  
            point = point.prev;  
    }  
}
```

**O(1)**

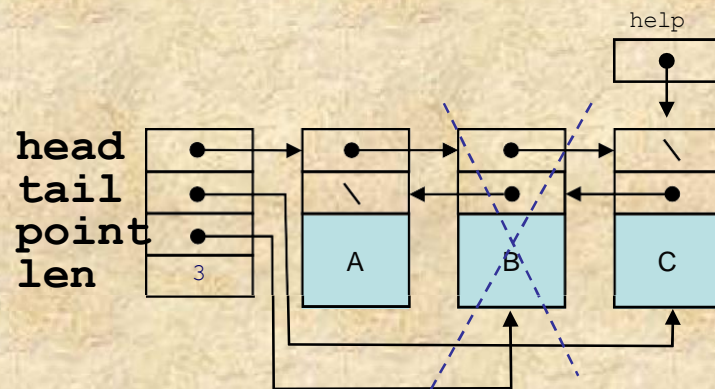
**prev** a **delete** posledního prvku  
jsou jediné operace, kde obousměrné  
zřetězený seznam sníží složitost



# Implementace seznamu



Obousměrně zřetězený



```
void delete( ) {
    Node help;
    if( point != null ){ // behind
        ignored
        help = point.next ;

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

        if( tail == point ) //last
            tail = tail.prev;

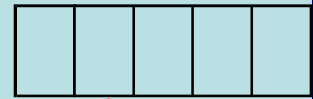
        if( help != null ) //update prev
            help.prev = point.prev;

        if( point.prev != null ); //upd
            next point.prev.next = help;

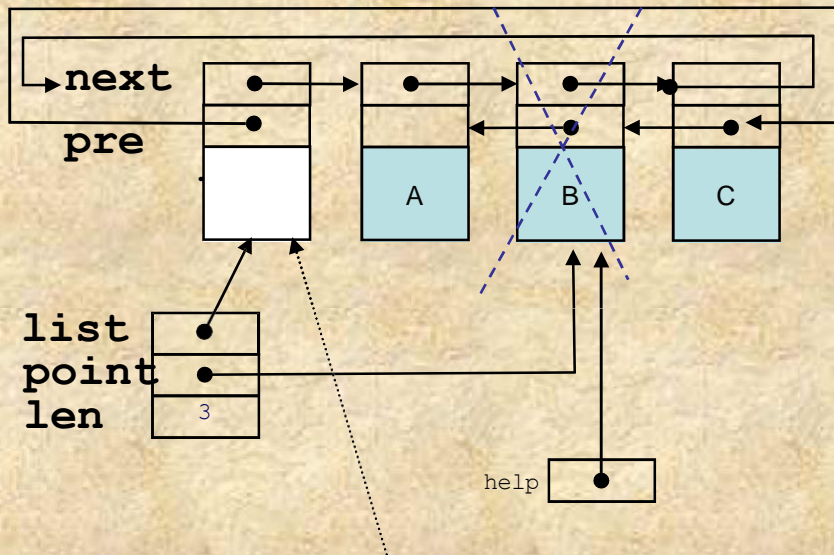
        point = help;

        len--;
    }
}
```

# Implementace seznamu



Kruhový obousměrně zřetězený



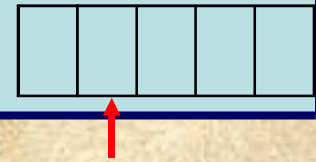
```
void delete( ) {  
    Node help;  
    if( point != list ){ //not empty  
        point.prev.next = point.next;  
        point.next.prev = point.prev;  
  
        help = point;  
        point = point.next;  
  
        len--;  
    }  
}  
  
void prev( ) {  
    if( !atBegin() ) //point != list.next  
        point = point.prev;  
}
```

## Konvence:

**Zarážka (dummy head)** - prvek navíc, který nenese data, zjednoduší operaci delete.

Ovlivní implementaci dalších operací!

## Shrnutí seznamu



- Jedna z nejzákladnějších DS ve výpočetní technice (používá se k implementaci jiných DS, stack, queue,...)
- Kontejner s rychlým přístupem ke všem prvkům bez upřednostnění konců
- Optimalizovaný pro vkládání a mazání prvků v libovolné pozici – v místě ukazovátka
- Nemá možnost indexovaného/přímého přístupu
  
- Lineární
- Homogenní
- Dynamický



# Reference

- 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/vvuka/upr/textv/adt/index.html>
- Aho, Hopcroft, Ullman: Data Structures and Algorithms, Addison-Wesley, 1987

# Reference

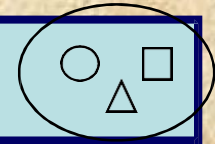
- Steven Skiena: The Algorithm Design Manual, Springer-Verlag New York, 1998  
<http://www.cs.sunysb.edu/~algorithm>
- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press, 1990
- Code examples: M.A.Weiss: Data Structures and Problem Solving using JAVA, Addison Wesley, 2001, code web 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](http://en.wikipedia.org/w/index.php?title=Abstract_data_type&oldid=78362071)

# Abstraktní datové typy

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



# Množina bez opakování (Set)



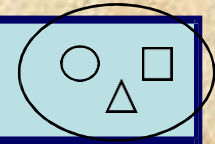
## Vlastnosti

- Množina je soubor objektů, chápaný jako celek
- Je jednoznačně určena svými prvky (ale neurčuje jejich pořadí ani žádné další vztahy)
- V podstatě je to tabulka (map), v níž položky mají pouze klíč
- Chybí asociovaná informace, zajímá nás pouze existence/neexistence prvku (klíče) v množině

## Použití

- Množina jako základní matematická struktura
- Může sloužit jako doplňující ADT v kombinaci s jiným typem
- Bitové masky (např. požadavky jednotlivých zařízení o přerušení, stavový registr v procesoru,...)

# Množina bez opakování (Set)



**Množina z pohledu matematika** = *neuspořádaný souhrn (kolekce) prvků* - bez opakovaných výskytů

Je plně určena

- **výčtem prvků**, např.  $\{1, 5, 99\}$  nebo  $\{1, 99, 5\}$  (nezáleží na pořadí)
- **definováním vlastností prvků**  $\{x \mid x \in N \wedge x \leq 100\}$
- $\emptyset = \Omega$  = **prázdná množina**

**Množina z pohledu programátora**

- **volba takové reprezentace**, aby se dobře implementovaly potřebné množinové operace
  - ⇒ **pole bitů (bitová mapa)**, nebo
  - ⇒ **seřazený seznam prvků dle relace (případného) uspořádání**

# Typy množin jako ADT

## Neuspořádaná množina

- Nejobecnější, nejbližší matematickému pojetí
- Neexistuje žádná relace uspořádání mezi prvky

## Částečně uspořádaná množina

- většina datových typů v počítačích
- relace částečného uspořádání:  $x \leq y$
- a navíc existují neporovnatelné prvky (neplatí ani  $x \leq y$  ani  $y \leq x$ )

### Relace uspořádání $\leq$ musí být

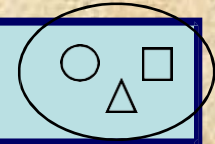
- **reflexivní**  $(x \leq x)$
- **antisymetrická**  $(x \leq y) \ \& \ (y \leq x) \Rightarrow x = y$
- **tranzitivní**  $(x \leq y) \ \& \ (y \leq z) \Rightarrow x \leq z$

## Úplně uspořádaná množina

- Pro libovolné dva prvky  $x, y \in S$  platí buď  $x \leq y$  nebo  $y \leq x$
- Např. typ `int`



# Možné operace nad množinou



## Konstruktor

- `init()` – vytvoří prázdnou množinu

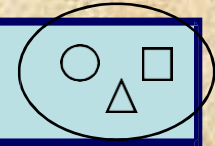
## Modifikující operace ( $x$ je typu `Key` a $S, A, B$ jsou množiny)

- `insert(S, x)` – vloží do množiny  $S$  prvek  $x$
- `delete(S, x)` – vyjme z množiny  $S$  prvek  $x$

## Predikáty (*queries*)

- `member(S, x)` – vrátí `true`, pokud  $x \in S$ , nebo `false`, pokud  $x \notin S$   
(je možná i jiná/šikovnější interpretace ...)
- `equal(A, B)` – vrátí `true`, pokud množiny obsahují stejné prvky
- `card(A)` – vrátí počet prvků množiny

# Možné operace nad množinou



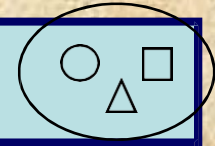
Pro úplně uspořádané množiny navíc operace

- **min**( $S$ ) – vrací nejmenší prvek
- **max**( $S$ ) – vrací největší prvek
- **succ**( $x, S$ ) – pro prvek  $x$  vrací nejbližší větší prvek, nebo **null**
- **pred**( $x, S$ ) – pro prvek  $x$  vrací nejbližší nižší prvek, nebo **null**

## Množinové operace

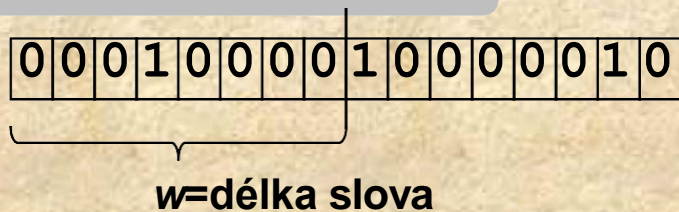
- **intersection**( $A, B$ ) – průnik množin  $A \cap B$
- **difference**( $A, B$ ) – rozdíl množin  $A - B$
- **union**( $A, B$ ) – sjednocení množin  $A \cup B$
- **merge**( $A, B$ ) – sjednocení disjunktních množin  $A \cup B$   
(musí být  $A \cap B \neq \emptyset$ )

# Implementace množiny



Implementace závisí na operacích a velikosti množiny

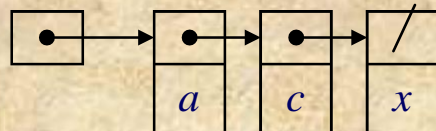
Bitovým vektorem



- pro malý rozsah prvků  $k = 1, \dots, N$ ,  
 $N$  malé, předem známé
- $i$ -tý bit je true, pokud je  $i$  prvkem množiny
- $O(1)$  member, insert, delete
- $O(N/w)$  union, intersection, difference

Seřazeným seznamem

head



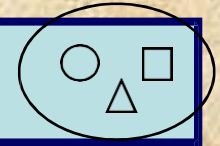
- pro libovolný rozsah možných prvků
- paměť úměrná počtu uložených prvků  $N$
- $O(N)$  member, insert, delete
- $O(N)$  union, intersection  
a difference

Binárním vyhledávacím stromem

- bude probráno později -  $O(\log N)$  až  $O(N)$



# Implementace bitovým vektorem



Konkrétní implementace závisí na možnosti implementace bitové mapy v daném programovacím jazyce a dostupnosti bitových operací na celá ( $w$ -bitová) slova.

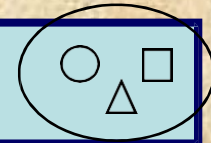
1 datový bit  $\approx$  1 bit paměti (výhody vs. nevýhody)

1 datový bit  $\approx$  1 byte paměti (výhody vs. nevýhody)

## Výpočetní složitost

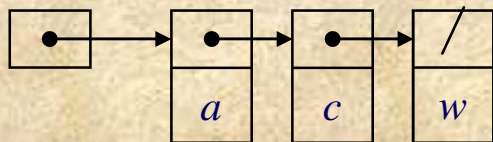
- `init` -  $O(N)$ , resp.  $O(N/w)$ ,  $N$  = velikost univerza
- `insert`, `delete`, `member` -  $O(1)$  !!!
- `equal`, `card`, `min`, `max`, `succ`, `pred` -  $O(N)$
- `intersection`, `union`, `difference`, `merge` -  $O(N)$ , resp.  $O(N/w)$ ,

# Implementace seřazeným seznamem

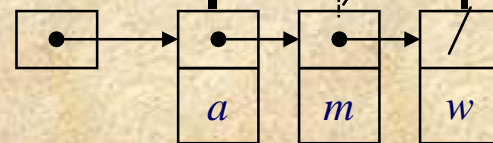


## Průnik

a.head



b.head



$O(m + n)$

### Průnik $a \cap b$

- porovnáváme prvky
- jsou shodné  $\Rightarrow$  na výstup, postup v obou
- nebo různé  $\Rightarrow$  postup v seznamu s menším prvkem

### Sjednocení $a \cup b$

- vložíme všechny v rostoucím pořadí
- jsou shodné  $\Rightarrow$  na výstup, postup v obou
- nebo různé  $\Rightarrow$  menší prvek na výstup a postup v seznamu s menším prvkem

### Rozdíl $a - b$

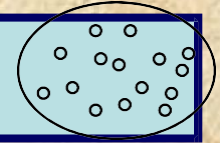
- shodné ignoruji
- různé
  - pokud menší  $a < b \Rightarrow a$  na výstup
  - pokud  $b < a$ , prvek v  $b$  přeskočím

## Abstraktní datové typy

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



# Množina s opakováním (Multiset)



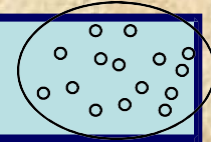
## Použití

- Všechny reálné množiny – aplikace se vstupy z reálných dat, kde se nedá zamezit opakování  
(databáze: Jiří Novák – 6x na FEL)
- Přidávají další atributy, aby se rozlišily

## Vlastnosti

- Rozšíření množiny o opakující se prvky

# Implementace množiny s opakováním



## Implementace

- bitovým vektorem **nejde** – musíme použít **vektor čítačů**
- uspořádaným seznamem - stejné jako u množiny bez opakování (povolí se opakování prvku / klíče)

## Problémy

- je třeba nově definovat přesnou sémantiku většiny operací
  - jak počítat počet prvků (započítáváme stejné NEBO jen různé ?)
  - jak sjednocení – počet stejných prvků je  $\max(m_i, n_i)$  nebo  $m_i+n_i$ ?
- jak velká počítadla budeme potřebovat?

## Množina - shrnutí

- Nelineární (asociativní) kontejner
  - pořadí uložených prvků není obecně definováno (z klíče nelze vypočítat adresa)
  - pro uspořádané klíče je pořadí dáno vzájemným porovnáním klíčů
- Homogenní – prvky stejného typu – klíč
- Dynamická

### Varianty

- Set - žádný prvek se neopakuje
- MultiSet - prvky se mohou opakovat



# Strom (Tree) a Graf (Graph)

**Těmto ADT se věnujeme později ...**

# 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/ad/index.html>
- Aho, Hopcroft, Ullman: Data Structures and Algorithms, Addison-Wesley, 1987



# References

- Steven Skiena: The Algorithm Design Manual, Springer-Verlag New York, 1998  
<http://www.cs.sunysb.edu/~algorithm>
- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press, 1990
- 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](#). 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](http://en.wikipedia.org/w/index.php?title=Abstract_data_type&oldid=78362071)