

Datové struktury a algoritmy

Část 11

Vyhledávání, zejména rozptylování

Petr Felkel

Topics

Vyhledávání

Rozptylování (hashing)

- Rozptylovací funkce
- Řešení kolizí
 - Zřetězené rozptylování
 - Otevřené rozptylování
 - Linear Probing
 - Double hashing

Slovník - Dictionary

Řada aplikací potřebuje

- dynamickou množinu
 - s operacemi: Search, Insert, Delete
- = **slovník**

Př. Tabulka symbolů překladače

identifikátor	typ	adresa
suma	int	0xFFFFDC09
...

Vyhledávání

Porovnáváním klíčů

$\Omega(\log n)$

- Nalezeno, když klíč_prvku = hledaný klíč
- např. sekvenční vyhledávání, BVS,...

asociativní

Indexováním klíčem (přímý přístup)

$\Theta(1)$

- klíč je přímo indexem (adresou)
- rozsah klíčů ~ rozsahu indexů

Rozptylováním

průměrně $\Theta(1)$

- výpočtem adresy z hodnoty klíče

adresní vyhledávání

Rozptylování - Hashing

= kompromis mezi rychlostí a spotřebou paměti

- ∞ času - sekvenční vyhledávání
- ∞ paměti - přímý přístup
 (indexování klíčem)
- málo času i paměti
 - hashing
 - velikost tabulky reguluje čas vyhledání

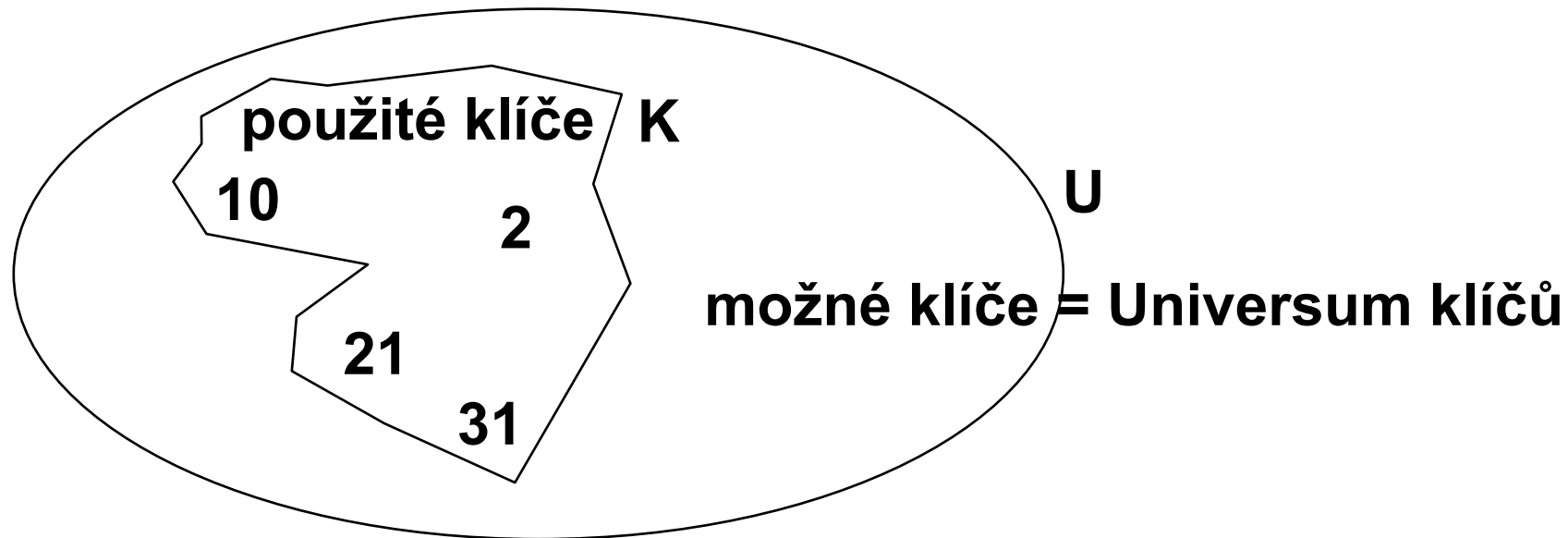
Rozptylování - Hashing

Konstantní očekávaný čas pro *vyhledání a vkládání* (*search and insert*) !!!

Něco za něco:

- čas provádění ~ délce klíče
- není vhodné pro operace *výběru podmnožiny a řazení* (*select a sort*)

Rozptylování

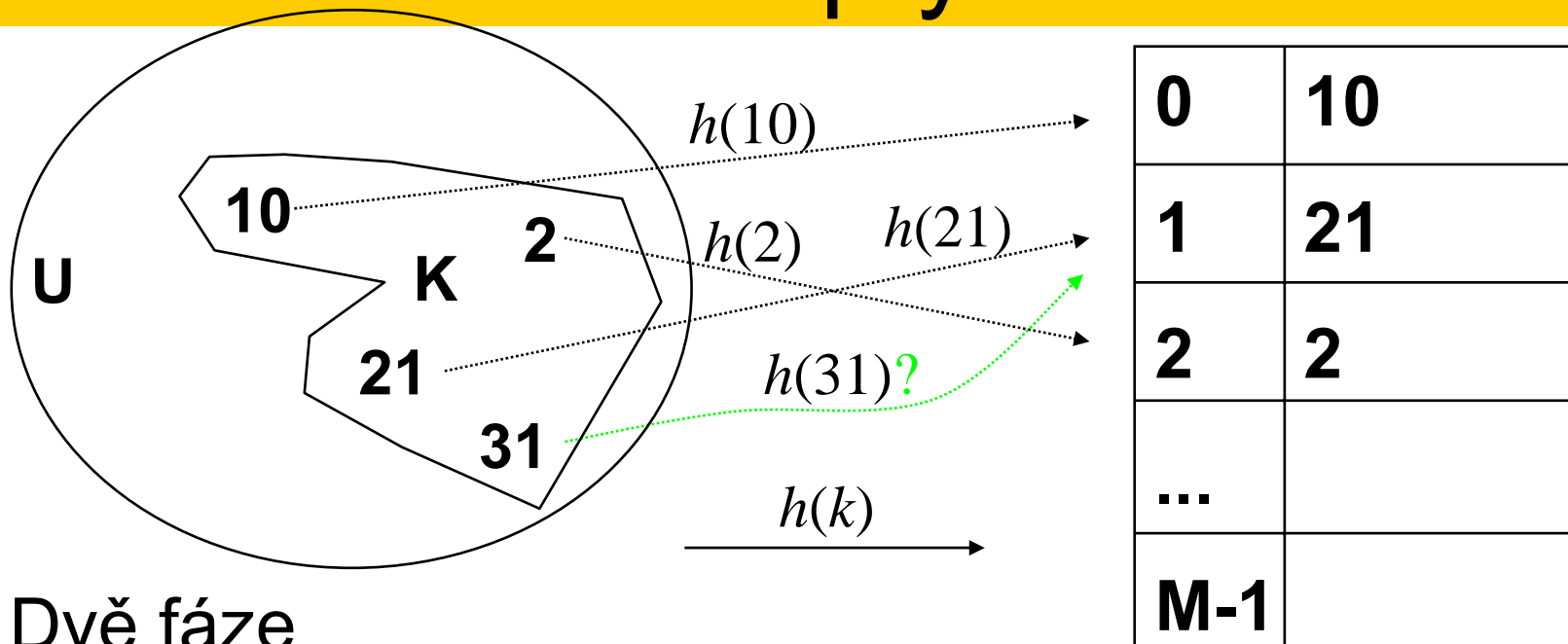


Rozptylování vhodné pro $|K| \ll |U|$

K množina použitých klíčů

U universum klíčů

Rozptylování



Dvě fáze

1. Výpočet rozptylovací funkce $h(k)$
($h(k)$ vypočítá adresu z hodnoty klíče)

2. Vyřešení kolizí

$h(31)$ **kolize**: index 1 již obsazen

1. Výpočet rozptylovací funkce $h(k)$

Rozptylovací funkce $h(k)$

Zobrazuje

množinu klíčů $K \in U$

do intervalu adres $A = \langle a_{min}, a_{max} \rangle$, obvykle $\langle 0, M-1 \rangle$

$|U| \gg |K| \cong |A|$

($h(k)$ Vypočítá adresu z hodnoty klíče)

Synonyma: $k_1 \neq k_2, h(k_1) = h(k_2)$
= kolize

Rozptylovací funkce $h(k)$

Je silně závislá na vlastnostech klíčů a jejich reprezentaci v paměti

Ideálně:

- výpočetně co nejjednodušší (rychlá)
- aproximuje náhodnou funkci
- využije **rovnoměrně** adresní prostor
- generuje **minimum kolizí**
- proto: využívá všechny složky klíče

Rozptylovací funkce $h(k)$

Každá hashovací funkce má slabá místa, kdy pro různé klíče dává stejnou adresu

Univerzální hashování

- Místo jedné hashovací funkce $h(k)$ máme konečnou množinu \mathbf{H} funkcí mapujících U do intervalu $\{0, 1, \dots, m-1\}$
- Při spuštění programu jednu náhodně zvolíme
- Tato množina je univerzální, pokud pro různé klíče $x, y \in U$ vrací stejnou adresu $h(x) = h(y)$ přesně v $|\mathbf{H}|/m$ případech
- Pravděpodobnost kolize při náhodném výběru funkce $h(k)$ je tedy přesně $1/m$

Rozptylovací funkce $h(k)$ - příklady

Příklady fce $h(k)$ pro různé typy klíčů

- reálná čísla
- celá čísla
- bitová
- řetězce

Chybná rozptylovací funkce

Rozptylovací funkce $h(k)$ -příklady

Pro **reálná čísla** 0..1

– multiplikativní: $h(k, M) = \text{round}(k * M)$

(neoddělí shluky blízkých čísel)

M = velikost tabulky (table size)

Rozptylovací funkce $h(k)$ -příklady

Pro **celá čísla** (w -bitová) - *for w -bit integers*

– multiplikativní: (kde M je prvočíslo)

- $h(k, M) = \text{round}(k / 2^w * M)$

– modulární:

- $h(k, M) = k \% M$

– kombinovaná:

- $h(k, M) = \text{round}(c * k) \% M, c \in \langle 0, 1 \rangle$

- $h(k, M) = (\text{int})(0.616161 * (\text{float}) k) \% M$

- $h(k, M) = (16161 * (\text{unsigned}) k) \% M$

Rozptylovací funkce $h(k)$ -příklady

Hash functions $h(k)$ - examples

Rychlá, silně závislá na reprezentaci klíčů

$h(k) = k \& (M-1)$ pro $M = 2^x$ (není prvočíslo),
a $\& =$ bitový součin

Rozptylovací funkce $h(k)$ -příklady

Pro řetězce (for *strings*):

```
int hash( char *k, int M )
{
    int h = 0, a = 127;
    for( ; *k != 0; k++ )
        h = ( a * h + *k ) % M;
    return h;
}
```

Hornerovo schéma: $k_2 * a^2 + k_1 * a^1 + k_0 * a^0 =$
 $((k_2 * a) + k_1) * a + k_0$

Rozptylovací funkce $h(k)$ -příklady

Pro řetězce: (pseudo-) randomizovaná

```
int hash( char *k, int M )
{
    int h = 0, a = 31415; b = 27183;
    for( ; *k != 0; k++, a = a*b % (M-1) )
        h = ( a * h + *k ) % M;
    return h;
}
```

Univerzální rozptylovací fce

- kolize s pravděpodobností $1/M$
- odlišná náhodná konstanta pro různé pozice v řetězci

Rozptylovací funkce $h(k)$ -chyba

Častá chyba: funkce *vrací stále stejnou hodnotu*

- chyba v konverzi typů
 - funguje, ale vrací blízké adresy
 - proto generuje hodně kolizí
- => *aplikace je extrémně pomalá*

Shrnutí

Rozptylovací funkce $h(k)$

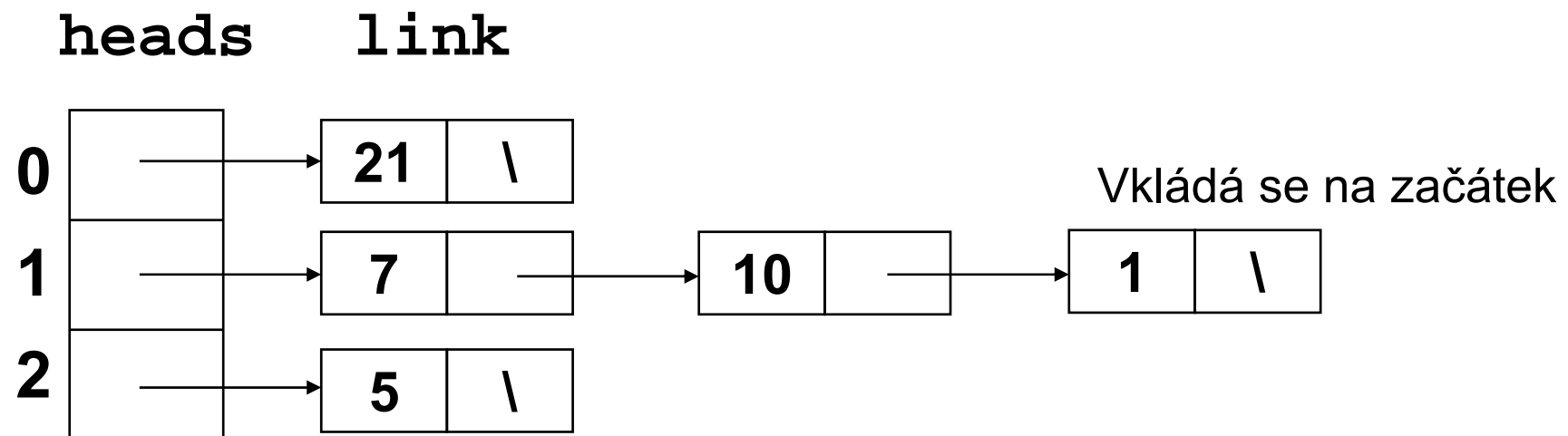
- počítá adresu z hodnoty klíče

2. Vyřešení kolizí

a) Zřetěžené rozptylování ^{1/5} Chaining

$$h(k) = k \bmod 3$$

posloupnost : 1, 5, 21, 10, 7



seznamy synonym

a) Zřetězené rozptylování 2/5

private:

```
link* heads; int N,M; [Sedgewick]
```

public:

```
init( int maxN )           // initialization
{
    N=0;                    // No.nodes
    M = maxN / 5;           // table size
    heads = new link[M];    // table with pointers
    for( int i = 0; i < M; i++ )
        heads[i] = null;
}
...
```

a) Zřetězené rozptylování 3/5

```
Item search( Key k )
{
    return searchList( heads[hash(k, M)], k );
}

void insert( Item item )                // Vkládá se na začátek
{
    int i = hash( item.key(), M );
    heads[i] = new node( item, heads[i] );
    N++;
}
```


a) Zřetězené rozptylování 4/5

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

Insert	$I(n) = t_{\text{hash}} + t_{\text{link}} = O(1)$	
Search	$Q(n) = t_{\text{hash}} + t_{\text{search}}$ $= t_{\text{hash}} + t_c * n/(2m) = O(n)$	<div>velmi nepravděpodobný extrém ↓ průměrně $O(1 + \alpha)$</div>
Delete	$D(n) = t_{\text{hash}} + t_{\text{search}} + t_{\text{link}} = O(n)$	$O(1 + \alpha)$

pro malá α (velká m) se hodně blíží $O(1)$!!!

pro velká α (malá m) m -násobné zrychlení x seq. s.

a) Zřetězené rozptylování 5/5

Praxe: volit $m = n/5$ až $n/10 \Rightarrow$ plnění $\alpha = 10$ prvků / řetěz

- vyplatí se hledání sekvenčně
- neplýtvá nepoužitými ukazateli

Shrnutí:

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

b) Otevřené rozptylování (open-address hashing)

Znám předem počet prvků (odhad)
nechci ukazatele (v prvcích ani tabulku)
=> posloupnost do pole

Podle tvaru hashovací funkce $h(k)$ při kolizi

1. lineární prohledávání (linear probing)
2. dvojí rozptylování (double hashing)

0	5
1	1
2	21
3	10
4	

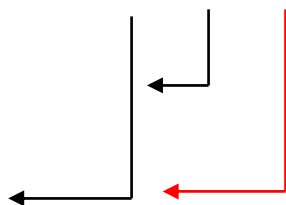
b) Otevřené rozptylování

$$h(k) = k \bmod 5$$

($h(k) = k \bmod m$, m je rozměr pole)

posloupnost: 1, 5, 21, 10, 7

0	5
1	1
2	
3	
4	



Problém:

kolize - 1 blokuje místo pro 21

1. linear probing

2. double hashing

Pozn.: 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

b) Otevřené rozptylování

(open-addressing hashing)

Metoda řešení kolizí
(solution of collisions)

b1) Linear probing

Lineární prohledávání

b2) Double hashing

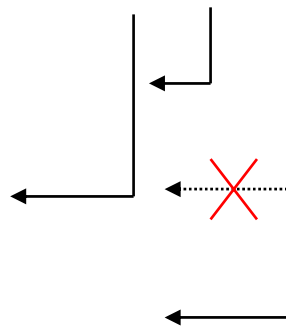
Dvojití rozptylování

b1) Linear probing

$$h(k) = [(k \bmod 5) + i] \bmod 5 = (k + i) \bmod 5$$

posloupnost: 1, 5, **21**, 10, 7

0	5
1	1
2	21
3	
4	



kolize - 1 blokuje

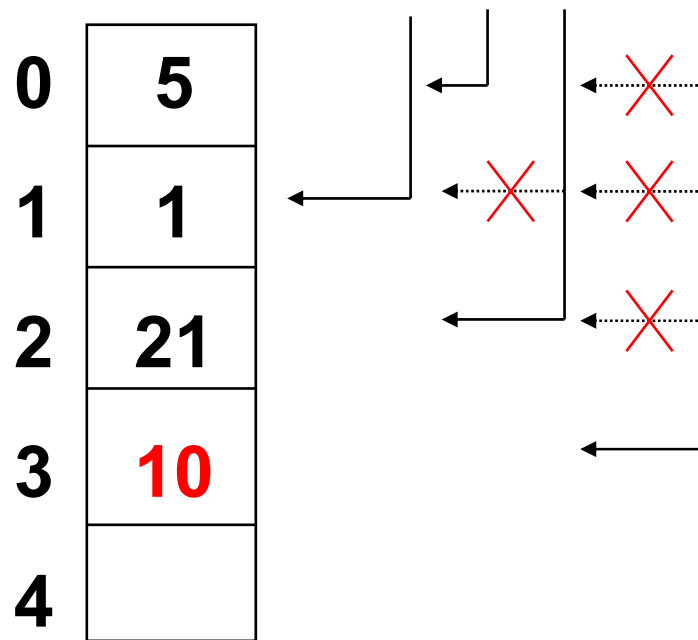
=> 1. linear probing

vlož o 1 pozici dál ($i++ \Rightarrow i = 1$)

b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

posloupnost: 1, 5, 21, **10**, 7



1. kolize - 5 blokuje - vlož dál

2. kolize - 1 blokuje - vlož dál

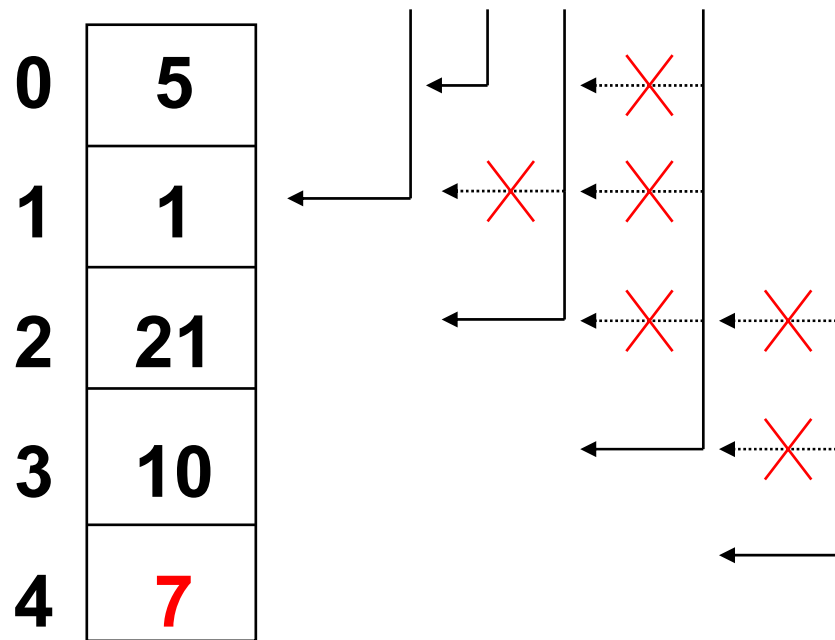
3. kolize - 21 blokuje - vlož dál

vloženo o 3 pozice dál ($i = 3$)

b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

posloupnost: 1, 5, 21, 10, **7**



1. kolize - vlož dál (i++)

2. kolize - vlož dál (i++)

vlož o 2 pozice dál (i = 2)

b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

posloupnost: 1, 5, 21, 10, 7

0	5	$i = 0$
1	1	$i = 0$
2	21	$i = 1$
3	10	$i = 3$
4	7	$i = 2$

b1) Linear probing

```
private:
    Item *st; int N,M;    [Sedgewick]
    Item nullItem;
public:
    init( int maxN )      // initialization
    {
        N=0;              // Number of stored items
        M = 2*maxN;        // load_factor < 1/2
        st = new Item[M];
        for( int i = 0; i < M; i++ )
            st[i] = nullItem;
    }...
```

b1) Linear probing

```
void insert( Item item )
{
    int i = hash( item.key(), M );

    while( !st[i].null() )
        i = (i+1) % M; // Linear probing

    st[i] = item;
    N++;
}
```

b1) Linear probing

```
Item search( Key k )
{
    int i = hash( k, M );

    while( !st[i].null() ) { // !cluster end
                            // zarážka (sentinel)
        if( k == st[i].key() )
            return st[i];
        else
            i = (i+1) % M; // Linear probing
    }
    return nullItem;
}
```

b) Otevřené rozptylování

(open-addressing hashing)

Metoda řešení kolizí
(solution of collisions)

b1) Linear probing

Lineární prohledávání

b2) Double hashing

Dvojití rozptylování

b2) Double hashing

Hash function $h(k) = [h_1(k) + i.h_2(k)] \bmod m$

$h_1(k) = k \bmod m$ // initial position
 $h_2(k) = 1 + (k \bmod m')$ // offset

} Both depend on k
=>

m = prime number or m = power of 2
 m' = slightly less m' = odd

Each key has
different
probe sequence

If d = greatest common divisor => search $1/d$ slots only

Ex: $k = 123456$, $m = 701$, $m' = 700$

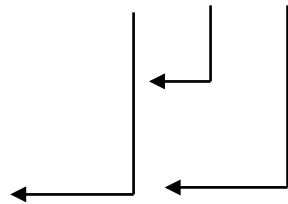
$h_1(k) = 80$, $h_2(k) = 257$ Starts at 80, and every 257 % 701

b2) Double hashing

$$h(k) = k \bmod 5$$

posloupnost: 1, 5, **21**, 10, 7

0	5
1	1
2	
3	
4	



kolize - 1 blokuje (collision-blocks)

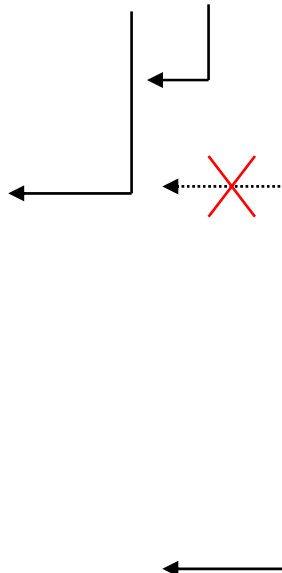
=> 2. double hashing

b2) Double hashing

$h(k) = [(k \bmod 5) + i \cdot h_2(k)] \bmod 5$, Nebo zjednodušeně (or simplified) $h(k) = (k + i \cdot 3) \bmod 5$
posloupnost: 1, 5, **21**, 10, 7

stačí prvočíslo $\neq m$

0	5
1	1
2	
3	
4	21



kolize - 1 blokuje
(collision blocks)

=> 2. double hashing

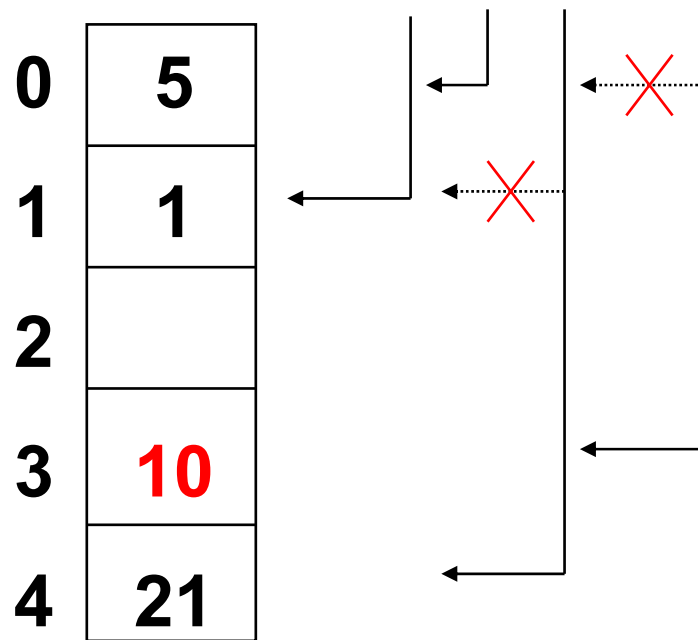
vlož o 3 pozice dál ($i++ \Rightarrow i = 1$)

(i je číslo pokusu)

b2) Double hashing

$h(k) = (k + i \cdot 3) \bmod 5$ (Př. velmi zjednodušené $h(k)$)

posloupnost: 1, 5, 21, **10**, 7



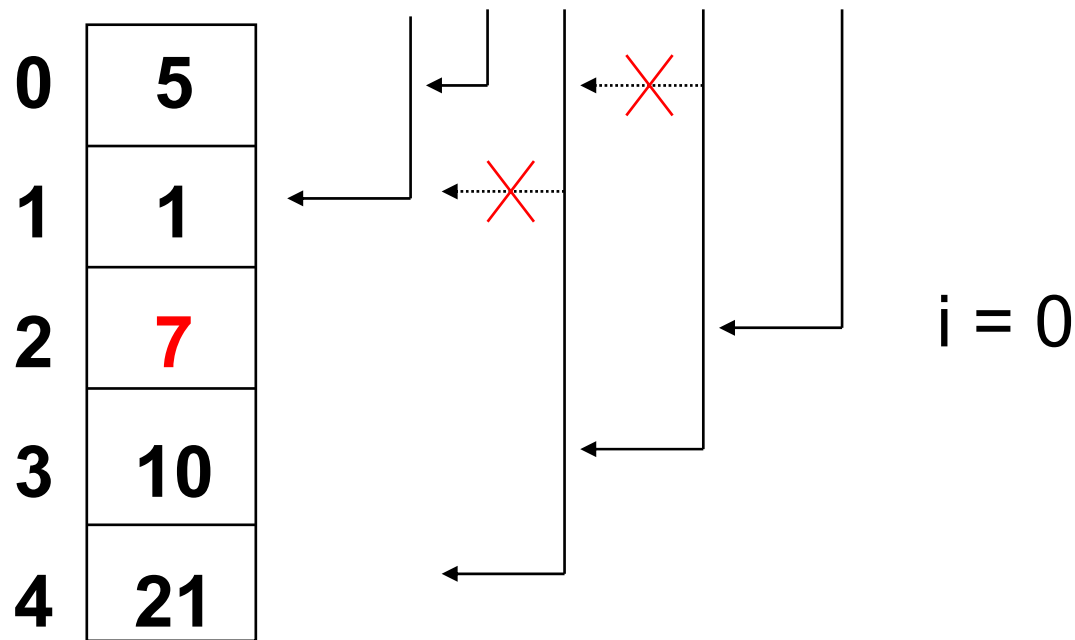
kolize - 5 blokuje - vlož dál

(vlož o 3 pozice dál ($i = 1$))

b2) Double hashing

$h(k) = (k + i \cdot 3) \bmod 5$ (Př. velmi zjednodušené $h(k)$)

posloupnost: 1, 5, 21, 10, 7



b2) Double hashing

$h(k) = (k + i \cdot 3) \bmod 5$ (Př. velmi zjednodušené $h(k)$)

posloupnost: 1, 5, 21, 10, 7

0	5	$i = 0$
1	1	$i = 0$
2	7	$i = 0$
3	10	$i = 1$
4	21	$i = 1$

Linear probing x Double hashing

$$h(k) = (k + i) \bmod 5$$

0	5	$i = 0$
1	1	$i = 0$
2	21	$i = 1$
3	10	$i = 3 !$
4	7	$i = 2$

dlouhé shluky
(long clusters)

$$h(k) = (k + i \cdot 3) \bmod 5$$

0	5	$i = 0$
1	1	$i = 0$
2	7	$i = 0$
3	10	$i = 1$
4	21	$i = 1$

vnořené sekvence
(mixed probe sequences)

b2) Double hashing

```
void insert( Item item )
{
    Key k = item.key();
    int i = hash( k, M ),
        j = hashTwo( k, M ); // different for  $k_1 \neq k_2$ 

    while( !st[i].null() )
        i = (i+j) % M; //Double Hashing

    st[i] = item; N++;
}
```

b2) Double hashing

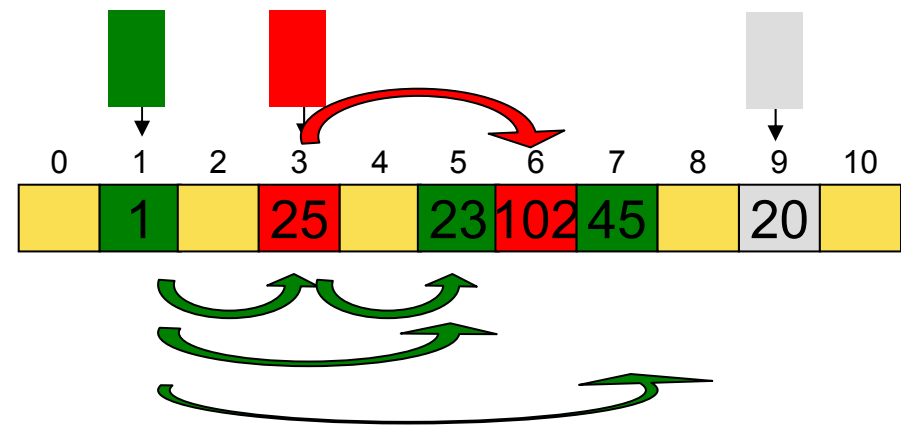
```
Item search( Key k )
{
    int i = hash( k, M ),
        j = hashTwo( k, M ); // different for  $k_1 \neq k_2$ 

    while( !st[i].null() )
    {
        if( k == st[i].key() )
            return st[i];
        else
            i = (i+j) % M; // Double Hashing
    }
    return nullItem;
}
```

Double hashing - example

b2) Double hashing $h(k) = [h_1(k) + i.h_2(k)] \bmod m$

Input	$h_1(k) = k \% 11$	$h_2(k) = 1 + k \% 10$	i	$h(k)$
1	1	2	0	1
25	3	6	0	3
23	1	4	0,1	1,5
45	1	6	0,1	1,7
102	3	3	0,1	3,6
20	9	1	0	9



$$h_1(k) = k \% 11$$

$$h_2(k) = 1 + (k \% 10)$$

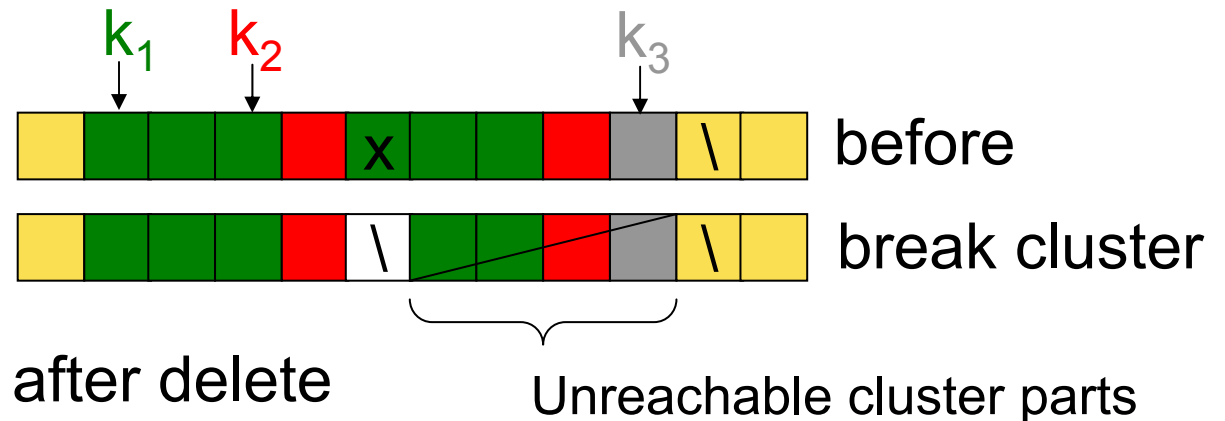
Item removal (delete)

Item 'x' removal

x replaced by null

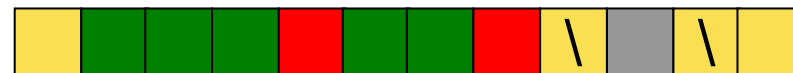
null breaks cluster(s) !!!

=> do not leave the hole after delete



Correction different for linear probing and double hashing

b1) in linear probing



=> **reinsert** the items after x (to the first null = to cluster end)

b2) in double hashing

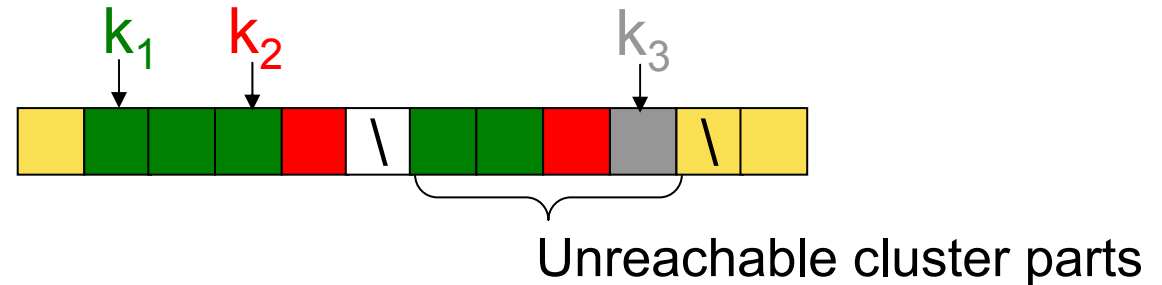


=> fill the hole up by a **special sentinel**

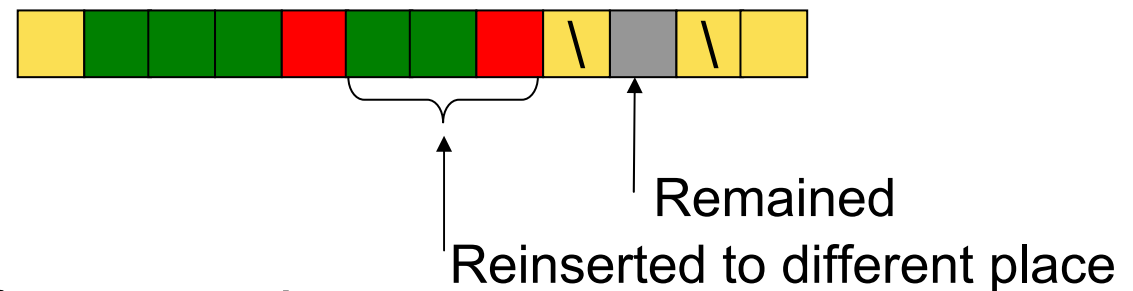
skipped by search, replaced by insert

Item removal (delete)

b1) in linear probing

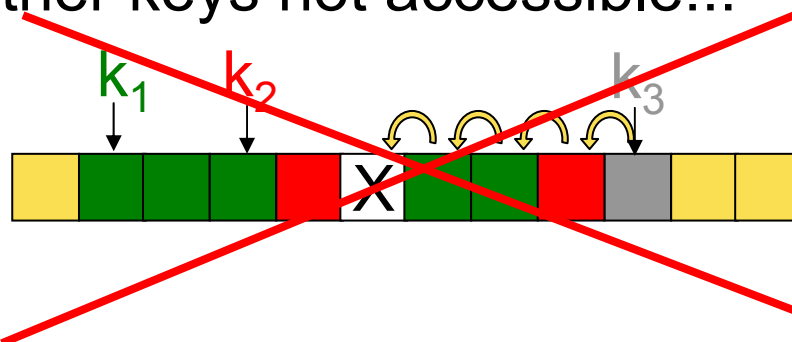


=> reinsert the items behind the cluster break (to the null)



=> avoid simple move of cluster tail

it can make other keys not accessible!!!

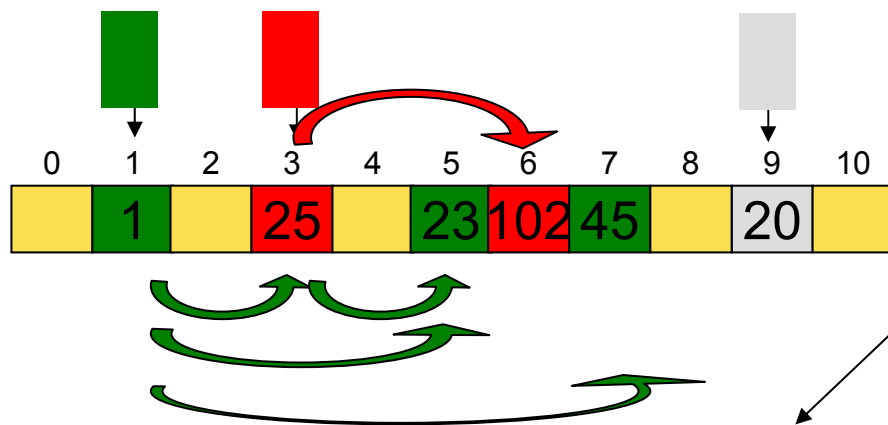


Linear-probing Item Removal

```
// do not leave the hole - can break a cluster
void remove( Item item )
{
    Key k = item.key();
    int i = hash( k, M ), j;
    while( !st[i].null() ) // find item to remove
        if( item.key() == st[i].key() ) break;
        else i = (i+1) % M;
    if( st[i].null() ) return; // not found
    st[i] = nullItem; N--; //delete,reinsert
    for(j = i+1; !st[j].null(); j=(j+1)%M, N--)
    {
        Item v = st[j]; st[j] = nullItem;
        insert(v); //reinsert elements after deleted
    }
}
```

Item removal (delete)

b2) Double hashing $h(k) = [h_1(k) + i.h_2(k)] \bmod m$



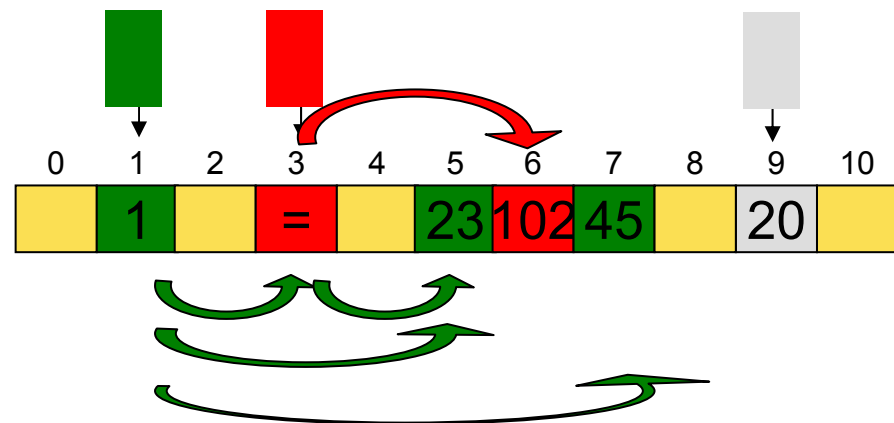
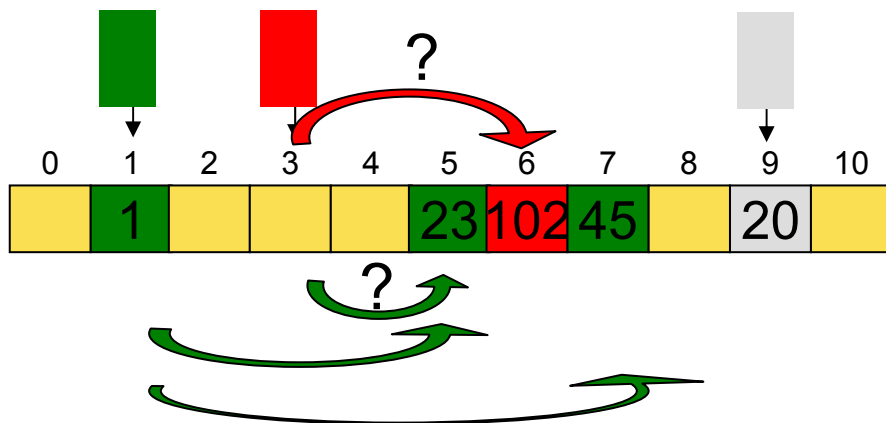
$$h_1(k) = k \% 11$$

$$h_2(k) = 1 + (k \% 10)$$

Remove 25

null – breaks paths to 23 and 102

Sentinel is correct



Double-hashing Item Removal

```
// Double Hashing - overlapping search sequences
//      - fill up the hole by sentinel
//      - skipped by search, replaced by insert
void remove( Item item )
{ Key k = item.key();
  int i = hash( k, M ), j = hashTwo( k, M );
  while( !st[i].null() ) // find item to remove
    if( item.key() == st[i].key() ) break;
    else i = (i+j) % M;
  if( st[i].null() ) return; // not found
  st[i] = sentinelItem; N--; // "delete" = replace
}
```

b) Otevřené rozptylování (open-addressing hashing)

α = plnění tabulky (*load factor of the table*)

$$\alpha = n/m, \alpha \in \langle 0, 1 \rangle$$

n = počet prvků (*number of items in the table*)

m = velikost tabulky, $m > n$ (*table size*)

b) Otevřené rozptylování (open-addressing hashing)

Average number of probes [Sedgewick]

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	$1 / (1 - \alpha)$

$$\alpha = n/m, \alpha \in \langle 0,1 \rangle$$

b) Očekávaný počet testů

Linear probing:

Plnění α	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 α	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

10 více testů

Tabulka může být více zaplněná než začne klesat výkonnost.
K dosažení stejného výkonu stačí menší tabulka.

References

[Cormen]

Cormen, Leiserson, Rivest: Introduction to Algorithms,
Chapter 12, McGraw Hill, 1990