Datové struktury a algoritmy

Část 11

Vyhledávání, zejména rozptylování

Petr Felkel

16.5.2016

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

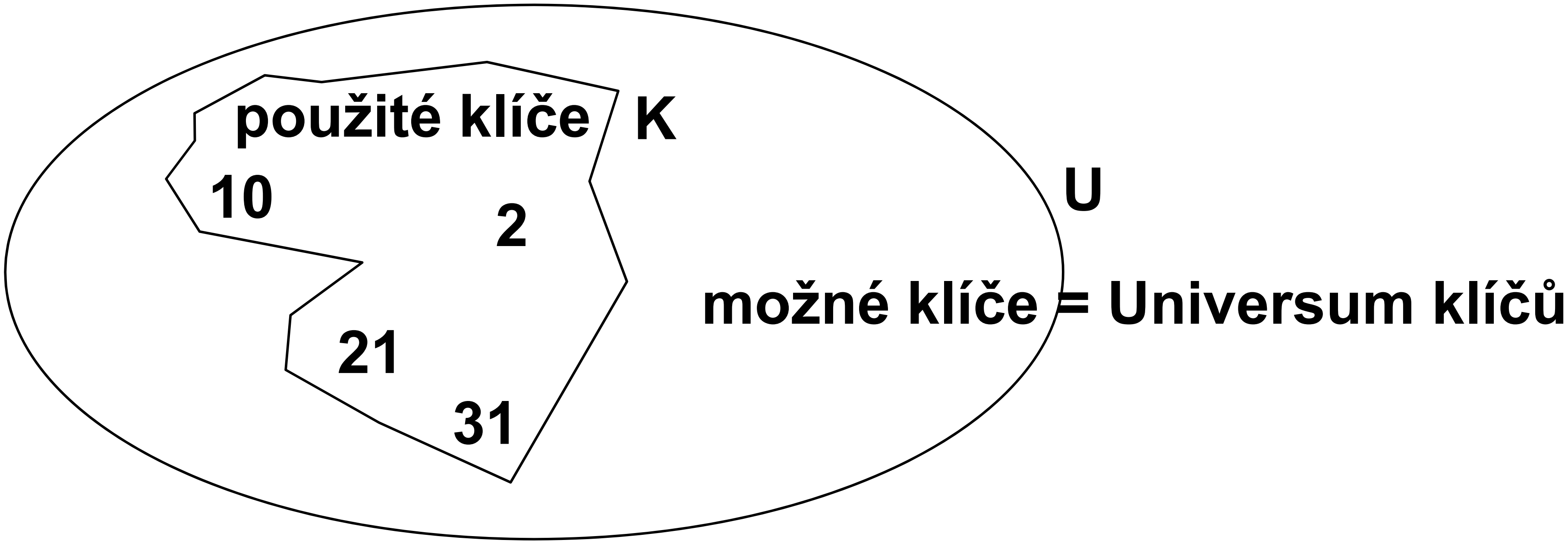
Rozptylování - Hashing

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

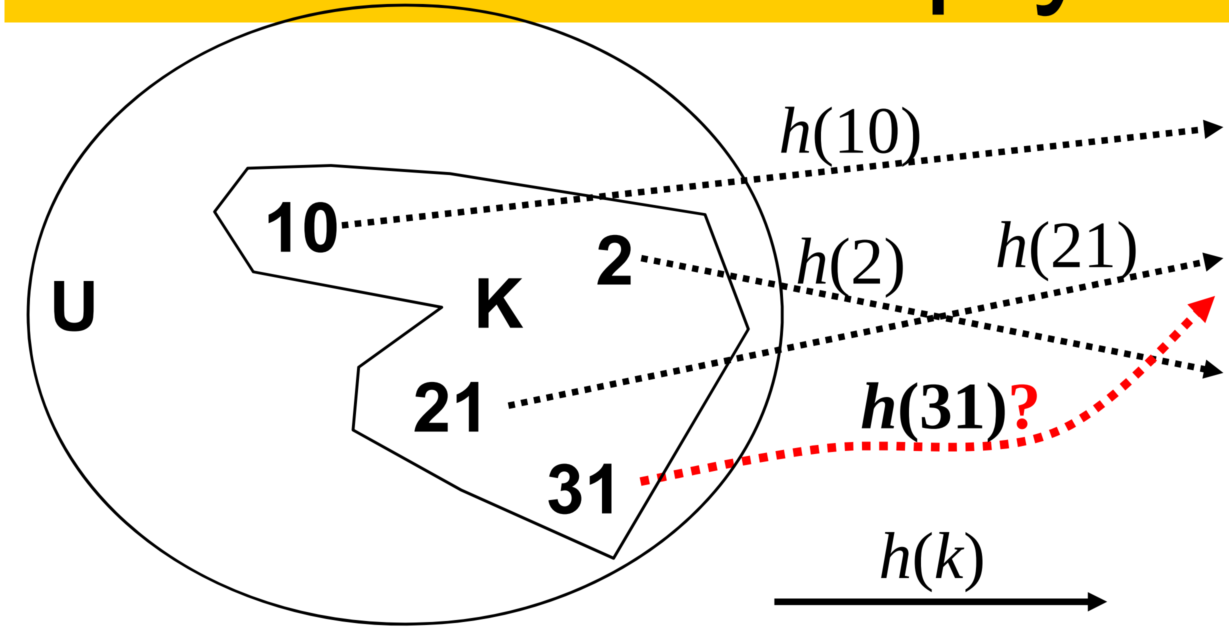


Rozptylování vhodné pro |K| << |U|

K množina použitých klíčů

U universum klíčů

Rozptylování



0	10
1	21
2	2
...	
M-1	

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 $A = <a_{min}, a_{max}>$, *obvykle* $<0, M-1>$

$|U| >> |K| \cong |A|$

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

Synonyma: $k_1 \neq k_2$, $\mathrm{h}(k_1) = \mathrm{h}(k_2)$

= 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*) - 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 z intervalu <0, 1>

– multiplikativní: **h(k,M) = round(k * M)**

neoddělí shluky blízkých čísel (s rozdílem < 1/**M**)

M = velikost tabulky (table size)

Rozptylovací funkce *h*(*k*)-příklady

Pro celá čísla

- multiplikativní: (kde M je prvočíslo, klíče mají w bitů)
  - **h(k,M) = round(k /2^w * M)**
- modulární:
  - **h(k,M) = k % M**
- kombinovaná:
  - **h(k,M) = round(c * k) % M, c $\in <0,1>$**
  - **h(k,M) = (int)(0.616161 * k) % M**
  - **h(k,M) = (16161 * k) % M** // pozor na přetečení

Rozptylovací funkce *h*(*k*)-příklady
Hash functions *h*(*k*) - examples

Rychlá, silně závislá na reprezentaci klíčů

h(k) = k & (M-1) **kde M = 2^x (není prvočíslo) ,**
& je bitový součin

je totéž jako

h(k) = k % M, tj.použije **x** nejnižších bitů klíče

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 :

$$P(a) = k_4 * a^4 + k_3 * a^3 + k_2 * a^2 + k_1 * a^1 + k_0 * a^0$$

$$= (((k_4 * a + k_3) * a + k_2) * a + k_1) * a + k_0$$

Výpočet hodnoty polynomu P v bodě a, koeficienty P jsou jednotlivé znaky (jejich číselná hodnota) v řetězci **`*k`**.

Rozptylovací funkce *h*(*k*)-příklady

Pro řetězce *(for strings) Java:*

```
public int hashCode( String s, int M ) {
        int h = 0;
        for( int i = 0; i < s.length(); i++ )
                h = 31 * h + s.charAt(i);
        return h;
 }
```

Hodnota konstant 127, 31 přispívá rovnoměrnému psoudonáhodnému rozptýlení.

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;
}
```

Rozptylovací funkce *h*(*k*)-chyba

Častá chyba:

funkce *vrací stále nebo většinou stejnou hodnotu*

– chyba v konverzi typů

– funguje, ale vrací blízké adresy

– proto generuje hodně kolizí

=> *aplikace je extrémně pomalá, řešení kolizí zdržuje.*

Shrnutí

Rozptylovací funkce h(k)

- počítá adresu z hodnoty klíče

Rozptylovací funkce $h(k)$

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

Univerzální hashování

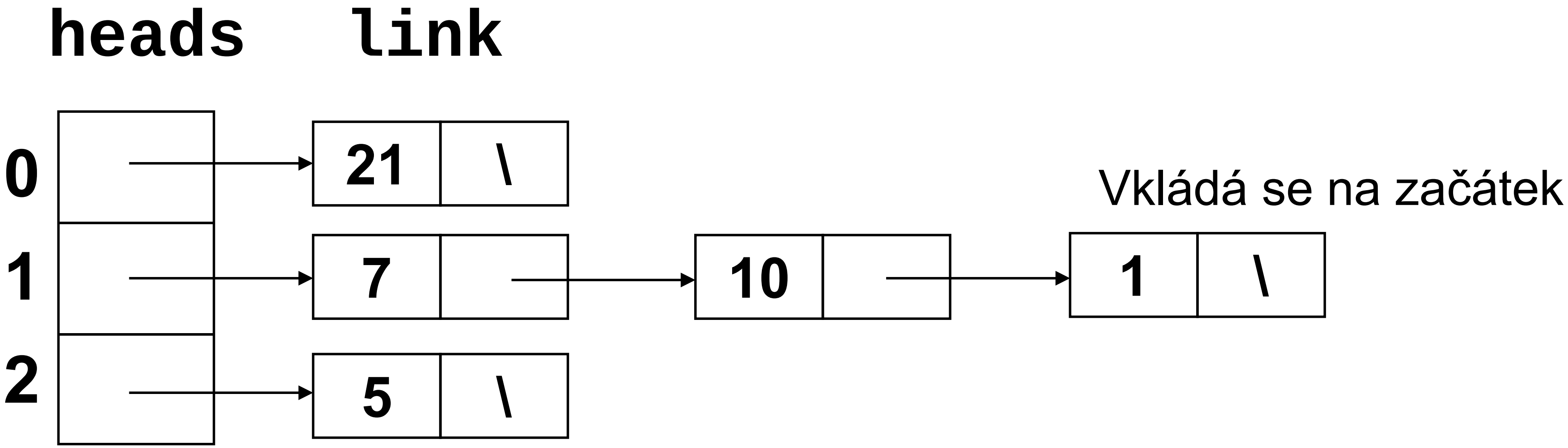
- Místo jedné hashovací funkce $h(k)$ máme konečnou množinu $\boldsymbol{H}$ funkcí mapujících U do intervalu $\{0, 1, \ldots, m-1\}$
- Při spuštění programu jednu náhodně zvolíme
- Tato množina je univerzální, pokud pro různé klíče $x,y \in U$ vrací stejnou adresu $h(x) = h(y)$ přesně v $|\boldsymbol{H}|/m$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

2. Vyřešení kolizí

a) Zřetězené 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. of 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

n = počet prvků, m = velikost tabulky, m<n.

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

velmi nepravděpodobný

extrém

průměrně

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

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

pro velká α (malá m) m-násobné zrychlení vůči sekvenčnímu hledání.

a) Zřetězené rozptylování 5/5

Praxe: volit *m* = *n*/5 až *n*/10 => plnění α = 10 prvků / řetěz

- vyplatí se hledání sekvenčně (je krátké)
- 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)

Známe předem počet prvků (odhad)

nechceme ukazatele (v prvcích ani tabulku)

=> posloupnost do pole

Podle tvaru hashovací funkce *h*(*k*) při kolizi:

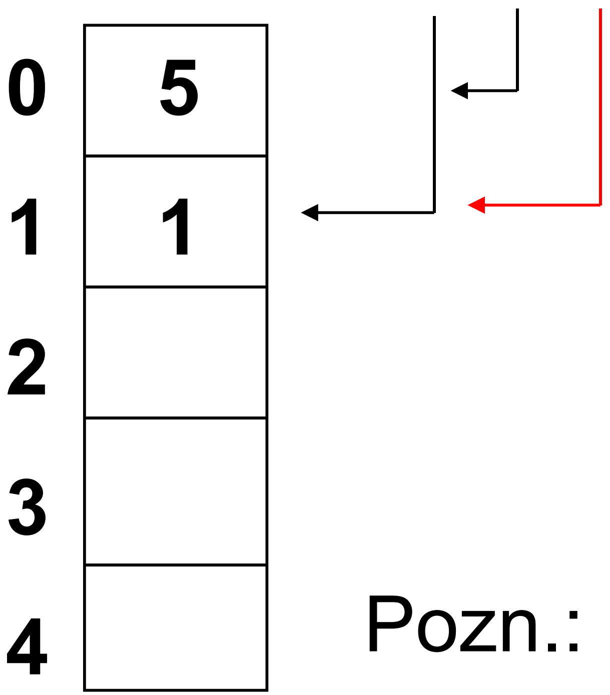
1. lineární prohledávání (linear probing)
2. dvojí rozptylování (double hashing)

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



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

Dvojí rozptylování

b1) Linear probing

$h(k) = [(k \bmod 5) + i\,] \bmod 5 = (k + i) \bmod 5;\ i = 0;$

posloupnost: 1, 5, 21, 10, 7

0	**5**
1	**1**
2	**21**
3	
4	

kolize - 1 blokuje

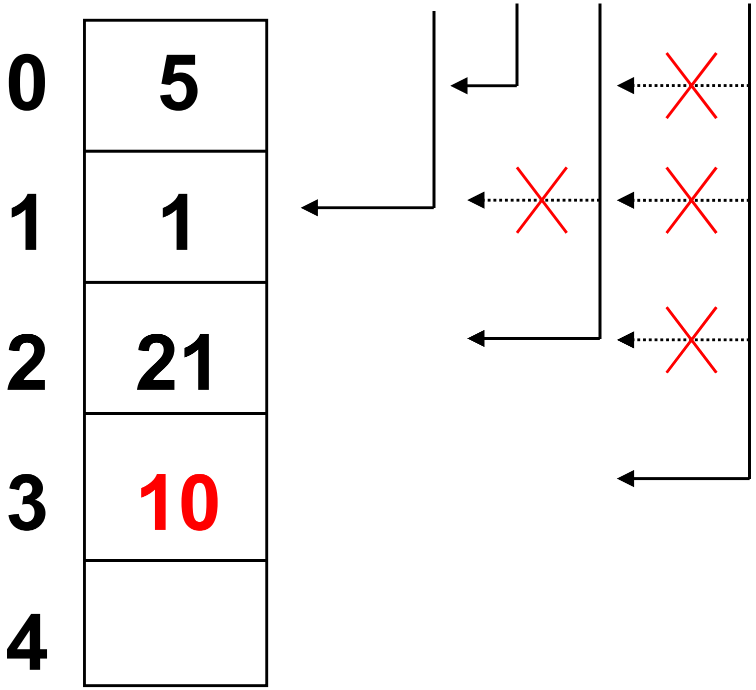
=> 1. linear probing

vlož o 1 pozici dál (i++ => i = 1)

b1) Linear probing

$h(k)$ = (k + i) mod 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

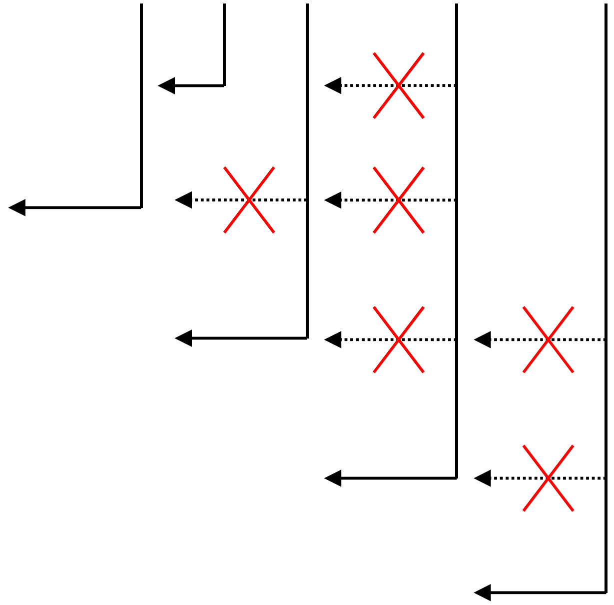
vloženo o 3 pozice dál (i = 3)

b1) Linear probing

h(*k*) = (k + i) mod 5

posloupnost: 1, 5, 21, 10, 7

0	**5**
1	**1**
2	**21**
3	**10**
4	**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

$h(k) = k \bmod 5$

posloupnost: 1, 5, 21, 10, 7

0	**5**
1	**1**
2	
3	
4	

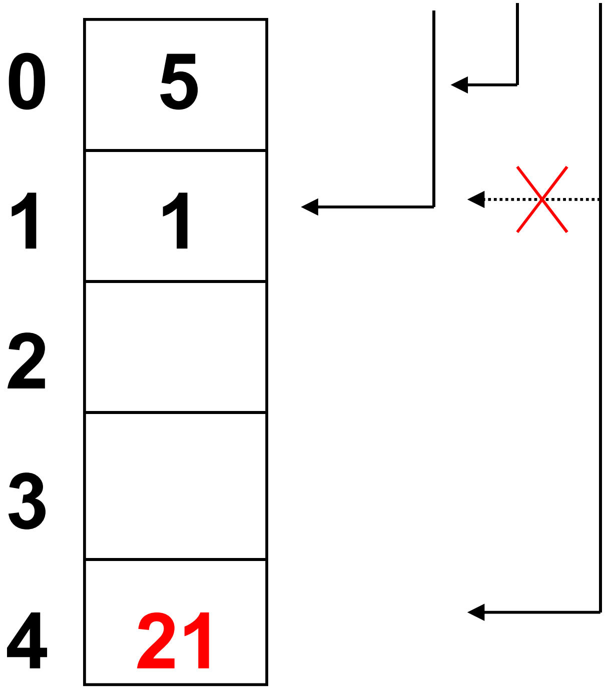
kolize - 1 blokuje (collision-blocks)

b1) Linear probing

$h(k)$ = [(k mod 5) + i.**const**] mod 5, $h(k)$ = (k + i.**3**) mod 5

posloupnost: 1, 5, 21, 10, 7

stačí prvočíslo ≠ m
nebo číslo nesoudělné s m



kolize - 1 blokuje

(collision blocks)

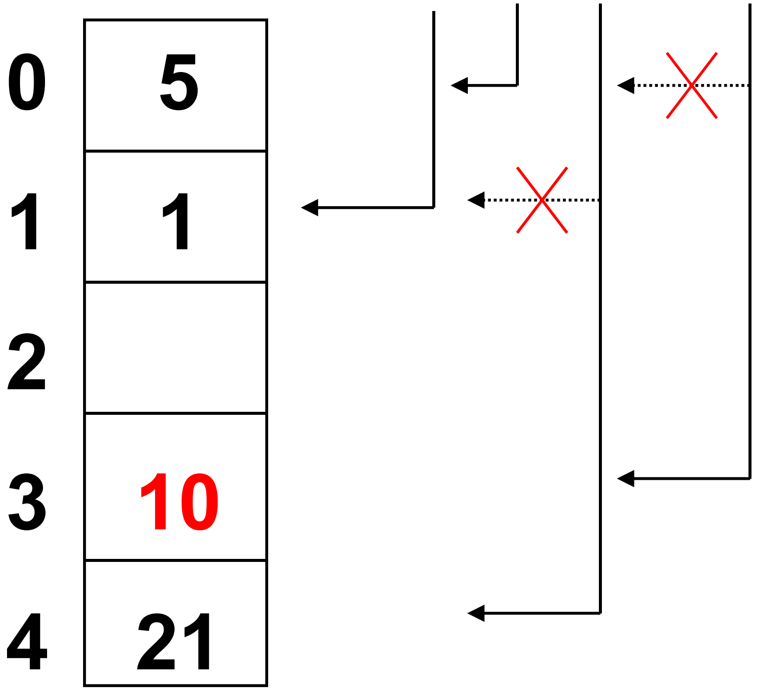
vlož o 3 pozice dál (i++ => i = 1)

(i je číslo pokusu)

b1) Linear probing

$h(k)$ = (k + i.3) mod 5

posloupnost: 1, 5, 21, 10, 7



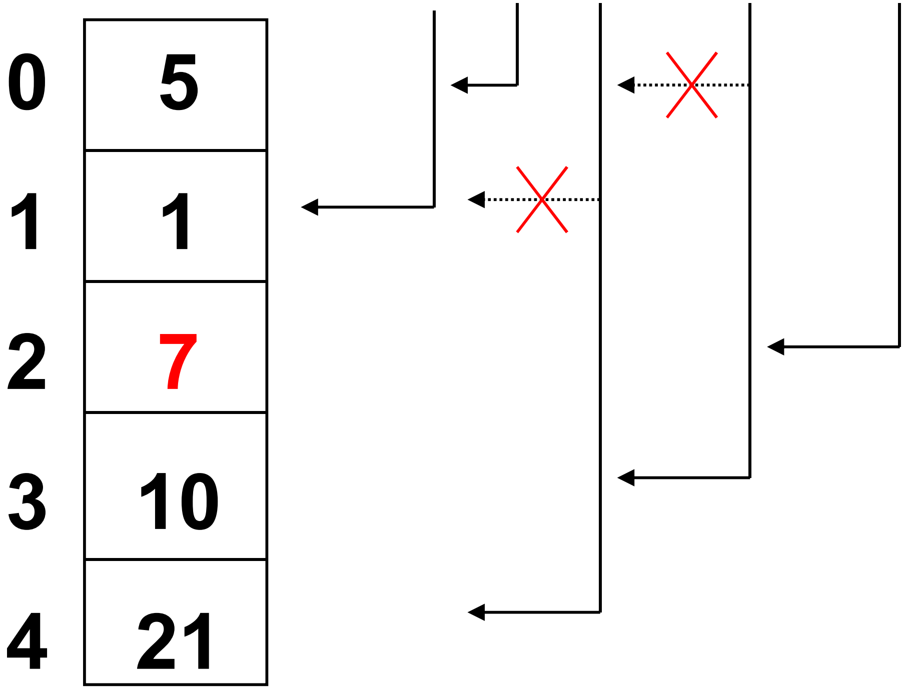
kolize - 5 blokuje - vlož dál

(vlož o 3 pozice dál (i = 1)

b1) Linear probing

$h(k)$ = (k + i.3) mod 5

posloupnost: 1, 5, 21, 10, 7



i = 0

b1) Linear probing

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

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

b1) Linear probing

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

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

hrozí dlouhé shluky
(long clusters)

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

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

vhodná volba posunu
i·3 je věcí náhody

b1) Linear probing

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

b1) Linear probing

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

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

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

b1) Linear probing

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

  while( !ht[i].null() ) { // !cluster end
                           // zarážka (sentinel)
     if( k == ht[i].key() )
          return ht[i];
     else
          i = (i+const) % 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

Dvojí 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

=>

Each key has different probe sequence

m = prime number or m = power of 2

m' = slightly less m' = odd

If d = greatest common divisor => search m/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

```
void insert( Item item )
{
  Key k = item.key();
  int i = hash( k, M ),
      j = hashTwo( k, M );// different for k1 != k2

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

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

b2) Double hashing

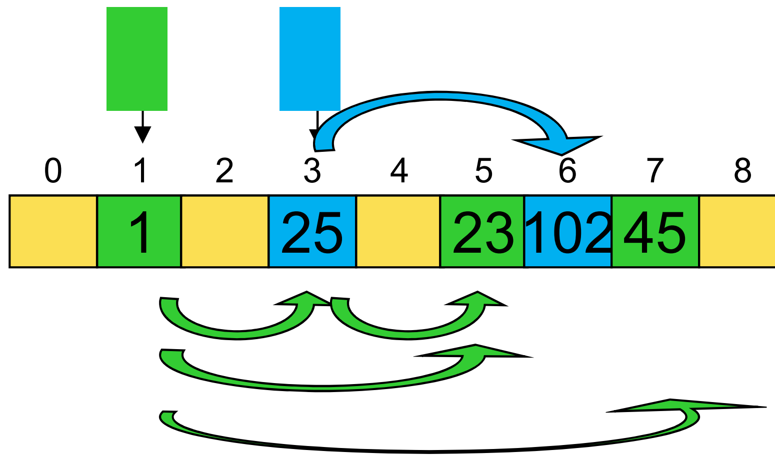
```
Item search( Key k )
{
  int i = hash( k, M ),
      j = hashTwo( k, M );// different for k1 != k2

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

Double hashing - example

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

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

b) Otevřené rozptylování (open-addressing hashing)

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

α **= 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)

Expected number of probes

Linear probing:

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

Double hashing:

Search hits	$(1 / \alpha) \ln \left(1 / (1 - \alpha) \right)$
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	2.0	3.0	4.0	10.0

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

or better:

[CLRS]

Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, third edition, Chapter 11, MIT press, 2009