Algoritmizace

Hashing I

Jiří Vyskočil, Marko Genyg-Berezovskyj

2010

Vyhledávání

- porovnáváním klíčů
  - např. sekvenční vyhledávání, vyhledávací stromy, …

$\Omega(\log n)$

- indexováním klíčem (s přímým přístupem)
  - klíč je zde přímo indexem (adresou)
  - rozsah klíčů odpovídá rozsahu indexů

$\Theta(1)$

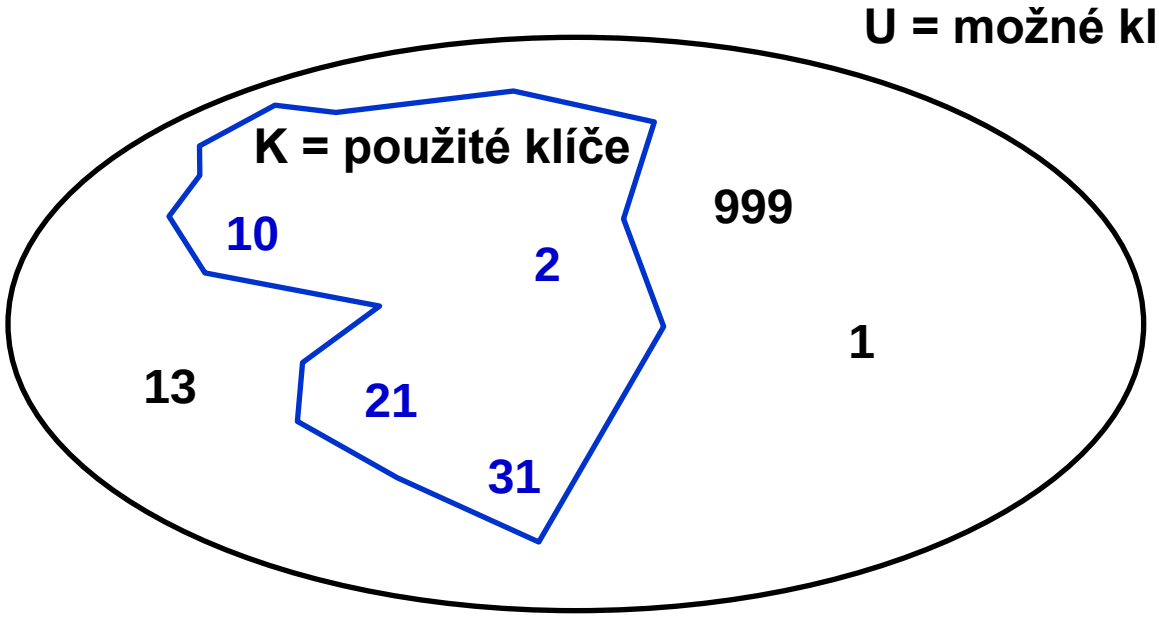
- hashováním (rozptylováním)
  - Index je získán výpočtem tzv. hashovací funkce z hodnoty klíče

průměrně $\Theta(1)$

Hashing (rozptylování)

- Největší úspora paměti: **sekvenční vyhledávání**
- Největší úsporu času: **indexování klíčem**
  - Často už dnes neplatí v souvislosti s moderním HW (cache paměti výrazně ovlivňují výsledek)
- **Hashování** je kompromis mezi rychlostí a spotřebou paměti.
  - velikost tabulky je určena oborem hodnot hashovací funkce
  - vhodný výběr hashovací funkce nám umožňuje docílit výsledků srovnatelných s indexováním klíčem při menší spotřebě paměti
  - očekávaný čas pro vkládání a vyhledávání je konstantní
  - v čase jedné operace je potřeba vždy alespoň spočítat hashovací funkci klíče

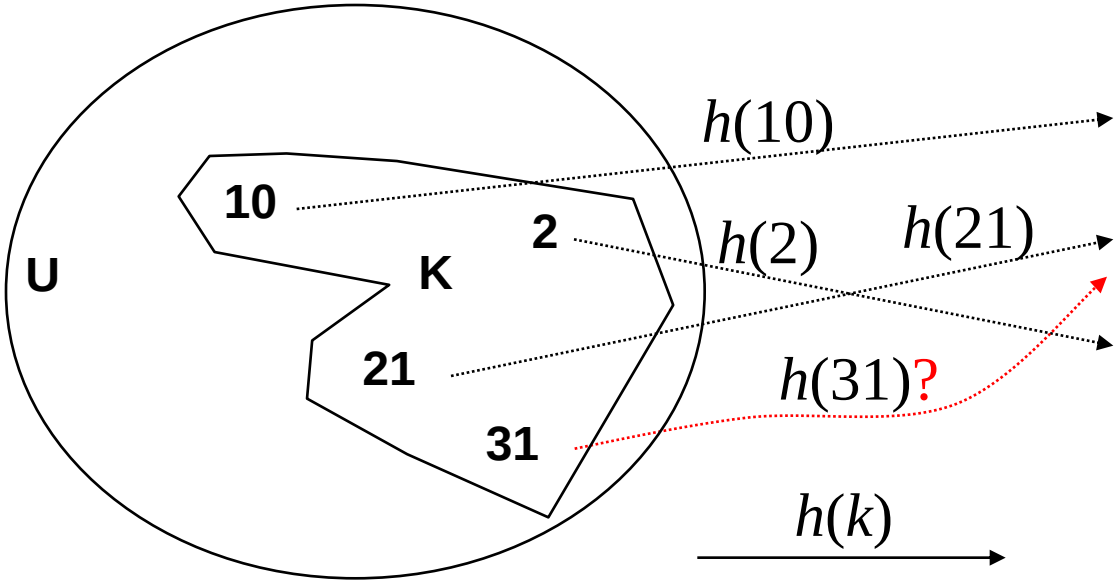
Hashing (rozptylování)



- Hashování je vhodné v případě, že **|K| << |U|**

Velikost množiny použitých klíčů je výrazně menší než velikost universa klíčů.

Hashing (rozptylování)



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

- proces hashování má dvě fáze
  1. Výpočet hashovací funkce $h(k)$
     ($h(k)$ vypočítá adresu z hodnoty klíče)
  2. Vyřešení kolizí
     $h(31)$ ..... kolize: index 1 již obsazen

Hashovací funkce $h(k)$

- Hashovací funkce $h(k)$ zobrazuje množinu klíčů $k \in \mathbf{U}$ do intervalu adres $A = \langle a_{min}, a_{max} \rangle$, obvykle $\langle 0, M-1 \rangle$
- $h(k)$ vypočítá adresu z hodnoty klíče
- Kolize vzniká v případě, kdy pro dva různé klíče vrací hashovací funkce h stejnou hodnotu

  tedy pro $k_1 \neq k_2$ platí, že $h(k_1) = h(k_2)$

Požadavky na hashovací funkci

- výpočetně je co nejjednodušší (rychlá)
- generuje minimum kolizí

- Důsledkem je, aby hashovací funkce
  - využívala rovnoměrně adresní prostor
  - aproximovala náhodnou funkci

- Proto se často pro výpočet hashovací funkce využívá celá hodnota klíče (resp. všechny složky klíče)

Příklady hashovacích funkcí

- **Základní metody**
  - **modulární metoda**

    $h(k) = k \mod m$

  - **multiplikativní metoda**

    A volíme z intervalu (0,1)

    $h(k) = \lfloor m (k * A \mod 1) \rfloor$

    Kde k A mod 1 znamená část za desetinnou tečkou, tedy $k * A - \lfloor k * A \rfloor$

Příklady hashovacích funkcí

- Pro **celá čísla** (w-bitová)
  - multiplikativní: (kde *m* je prvočíslo)
    - $h(k) = \text{round}(k / 2^w * m)$
  - modulární:
    - $h(k) = k \bmod m$
  - kombinovaná:
    - $h(k) = \text{round}(c * k) \bmod m$, $c \in \langle 0,1 \rangle$
    - $h(k)$ = (int)(0.616161 * (float) k) mod m
    - $h(k)$ = (16161 * (unsigned) k) mod m

Příklady hashovacích funkcí

- Pro **řetězce** *(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:** $k_2 * a^2 + k_1 * a^1 + k_0 * a^0 =$

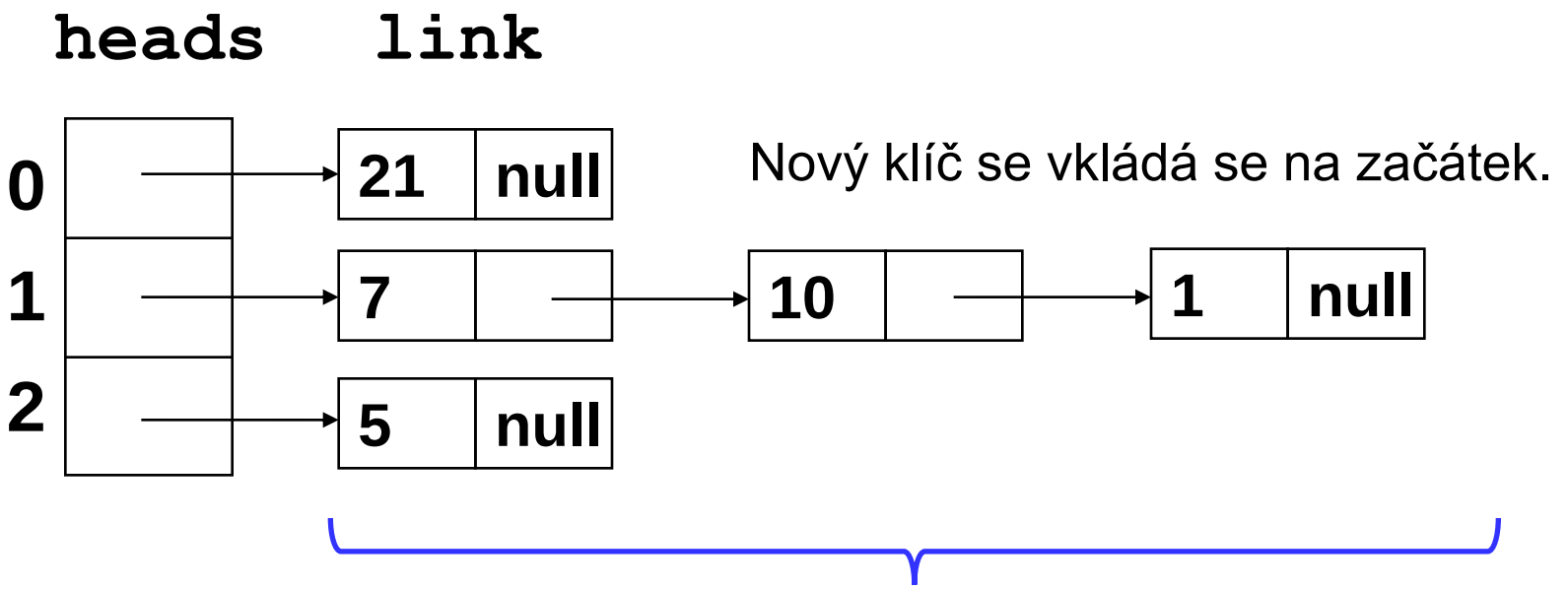
  $((k_2 * a) + k_1) * a + k_0$

Řešení kolizí

- perfektní
  - Hashovací funkce je zvolena tak, že kolize nikdy nenastane.
  - Je potřeba dopředu znát všechny použité klíče.
  - Hledání vhodné hashovací funkce může být velmi obtížné v závislosti na počtu použitých klíčů a zadané velikosti *m*.
- hashování se separovanými řetězci (chaining)
- otevřené hashování (open-address hashing)
  - lineární přidávání
  - dvojité hashovaní
- srůstající hashování (coalesced hashing)
- univerzální hashování
- dynamické hashování

hashování se separovanými řetězci

- $h(k) = k \bmod 3$
- posloupnost : 1, 5, 21, 10, 7



hashování se separovanými řetězci

- Řetěz synonym má v ideálním případě všude délku

  $\alpha = n/m$, $\alpha > 1$ (plnění tabulky)

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

Tento nejhorší případ, který je velmi nepravděpodobný!

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

- pro malá α (tedy velká m) se složitost blíží O(1) !!!
- pro velká α (tedy malá m) dochází k m-násobnému zrychlení oproti sekvenčnímu vyhledávání.

hashování se separovanými řetězci

- **praxe**
  - **volit *m* = *n*/5 až *n*/10 => plnění α = 10 prvků / řetěz**
  - vyplatí se hledání sekvenčně pro relativně krátké sekvence
  - neplýtvá nepoužitými ukazateli

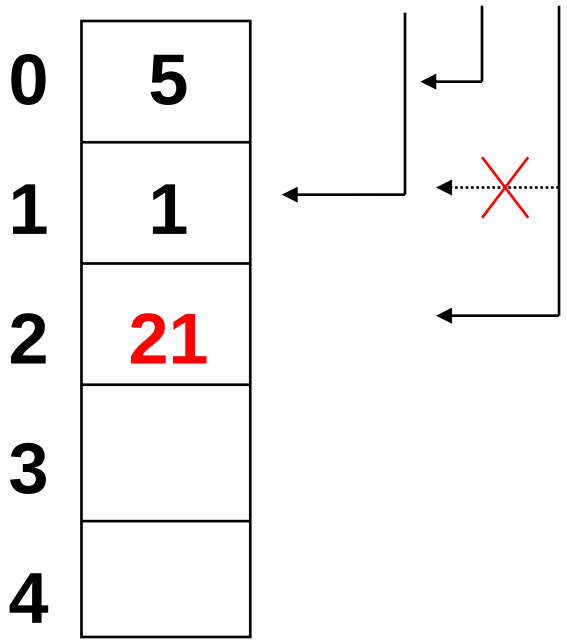
- **shrnutí**
  + nemusíme znát *n* předem
  – potřebuje dynamické přidělování paměti
  – potřebuje navíc paměť na ukazatele a na tabulku[*m*]
- **možná vylepšení**
  - Nahradit spojový seznam jinou datovou strukturou jako např.: vyhledávací strom, uspořádaným seznam, atd.

Otevřené hashování (open-address hashing)

- Znám předem počet prvků (odhad)
- Z důvodů efektivity nechci ukazatele (mezi prvky a ani v tabulce)
- => kolizní prvky (synonyma) se musejí „nějak" ukládat přímo do hashovací tabulky
  - lineární přidávání (linear probing)
  - dvojité hashovaní (double hashing)

Otevřené hashování s lineárním přidáváním

- $h(k) = [(k \bmod 5) + i] \bmod 5 = (k + i) \bmod 5$
- posloupnost: 1, 5, 21, 10, 7



kolize - 1 blokuje

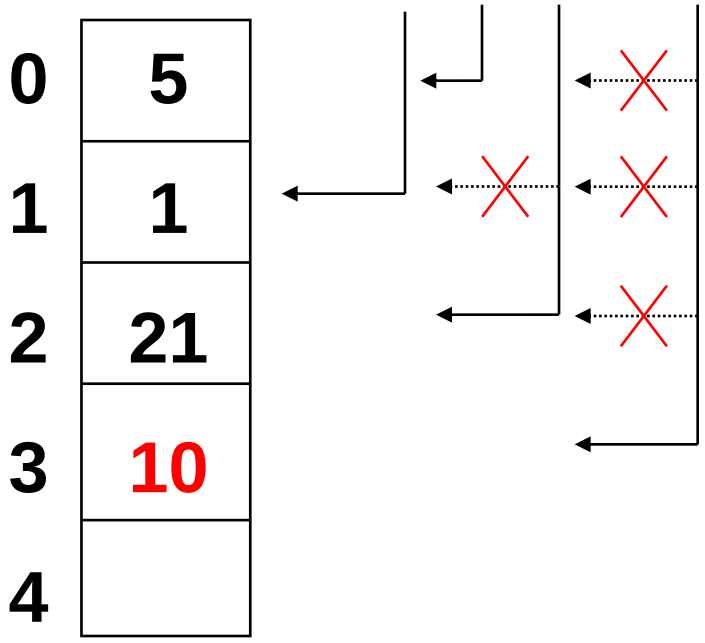
⇒Lineární přidávání (linear probing)

vlož o 1 pozici dál (i++ => i = 1)

Pokud není volná, postup opakuj.

Otevřené hashování s lineárním přidáváním

- $h(k) = (k + i) \bmod 5$
- posloupnost: 1, 5, 21, 10, 7



1. kolize - 5 blokuje - vlož dál

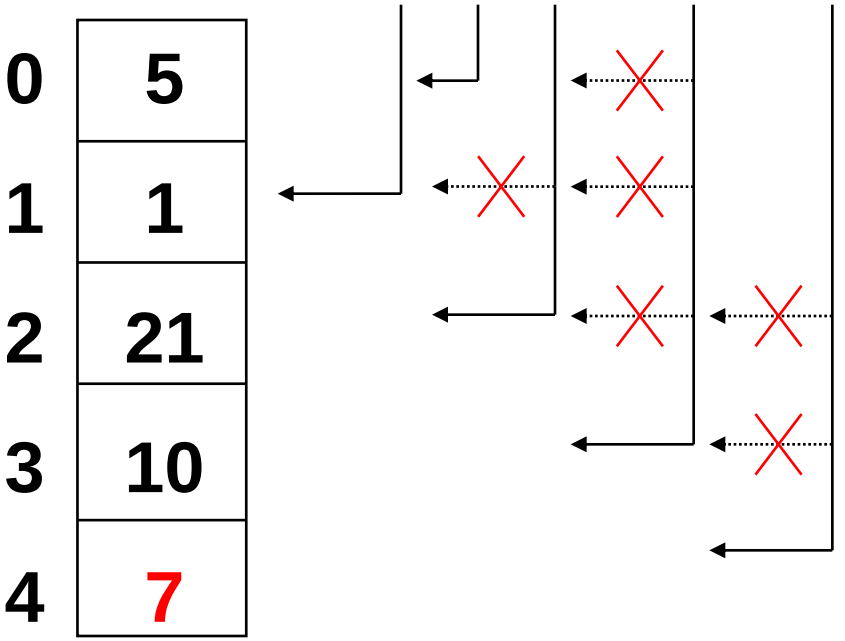
2. kolize - 1 blokuje - vlož dál

3. kolize - 21 blokuje - vlož dál

vloženo o 3 pozice dál (i = 3)

Otevřené hashování s lineárním přidáváním

- $h(k) = (k + i) \bmod 5$
- posloupnost: 1, 5, 21, 10, 7



1. kolize - vlož dál (i++)

2. kolize - vlož dál (i++)

vlož o 2 pozice dál (i = 2)

Otevřené hashování s lineárním přidáváním

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

  while( !st[i].null() ) { // opakuj dokud není konec
                           // clusteru (do zarážky)
      if( k == st[i].key() )
            return st[i];
      else
            i = (i+1) % M; // Linear probing
    }
      return nullItem;
}
```

Otevřené hashování s dvojitým hashováním

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

  kde $h_1(k) = k \bmod m$ // initial position

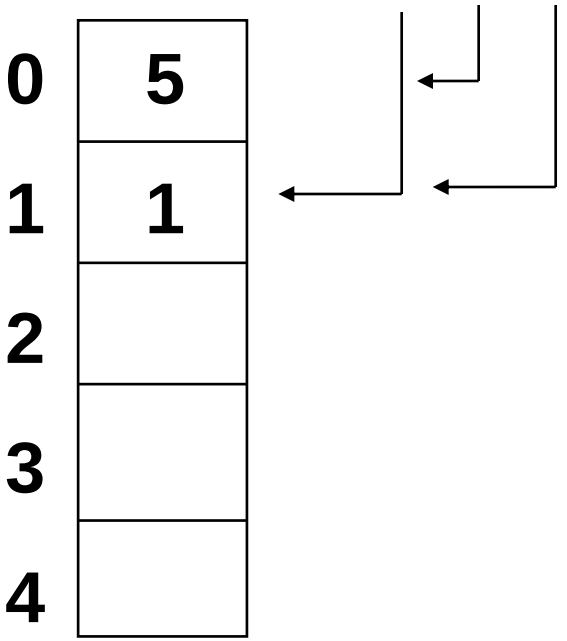
  $h_2(k) = 1 + (k \bmod m')$ // offset

  volíme m a m' jako různá prvočísla nebo

  m' je výrazně menší než m

Otevřené hashování s dvojitým hashováním

- $h(k) = k \bmod 5$
- posloupnost: 1, 5, 21, 10, 7



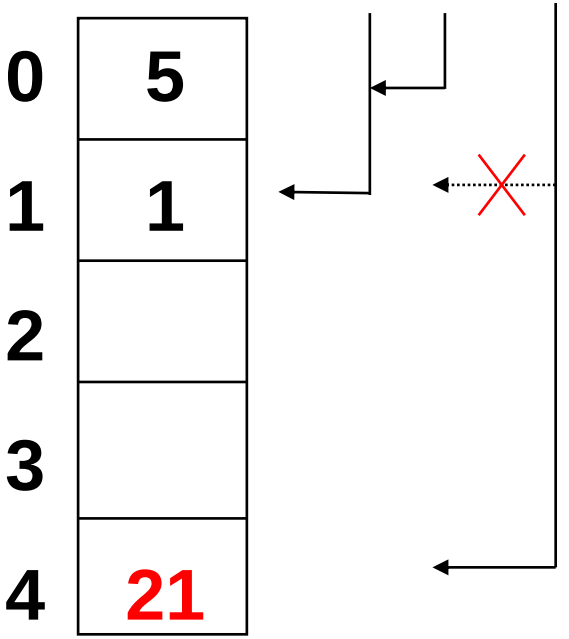
kolize - 1 blokuje (collision-blocks)

=> double hashing

Otevřené hashování s dvojitým hashováním

- $h(k) = [(k \bmod 5) + i.h_2(k)] \bmod 5 \Rightarrow h(k) = (k + i.3) \bmod 5$
- posloupnost: 1, 5, 21, 10, 7

stačí prvočíslo ≠ m



kolize - 1 blokuje

(collision blocks)

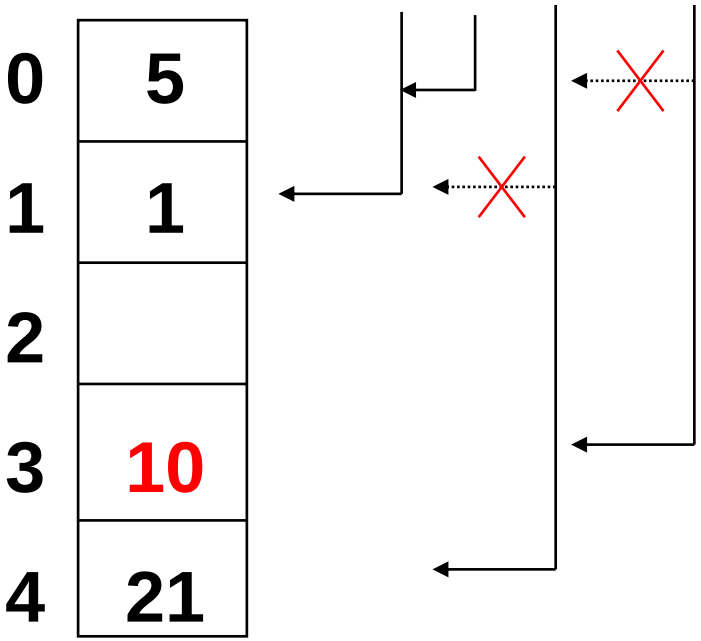
=> double hashing

vlož o 3 pozice dál (i++ => i = 1)

(i je číslo pokusu)

Otevřené hashování s dvojitým hashováním

- $h(k)$ = (k + i.3) mod 5 (Příklad velmi zjednodušené $h(k)$)
- posloupnost: 1, 5, 21, 10, 7

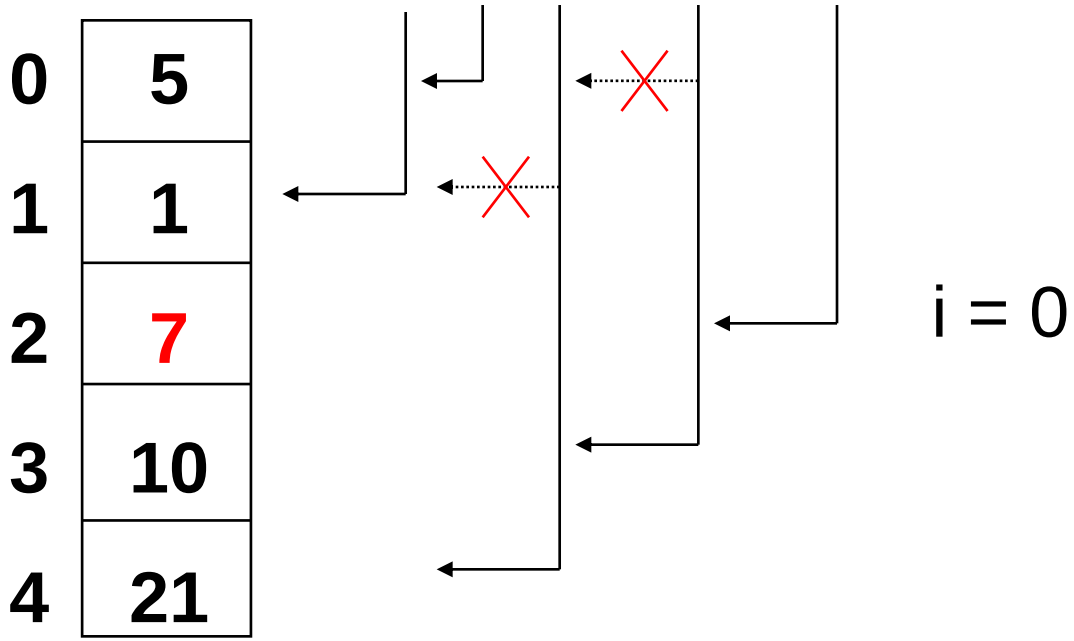


kolize - 5 blokuje - vlož dál

(vlož o 3 pozice dál (i = 1)

Otevřené hashování s dvojitým hashováním

- $h(k)$ = (k + i.3) mod 5 (Příklad velmi zjednodušené $h(k)$)
- posloupnost: 1, 5, 21, 10, 7



Otevřené hashování s dvojitým hashováním

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

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

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

Otevřené hashování s dvojitým hashováním

```
Item search( Key k )
{
  int i = hash( k, M ),
      j = hashTwo( k, M );// j se obvykle liší pro k1 != k2

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

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

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

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

Problémy: dlouhé shluky (long clusters)

vnořené sekvence (mixed probe sequences)

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

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