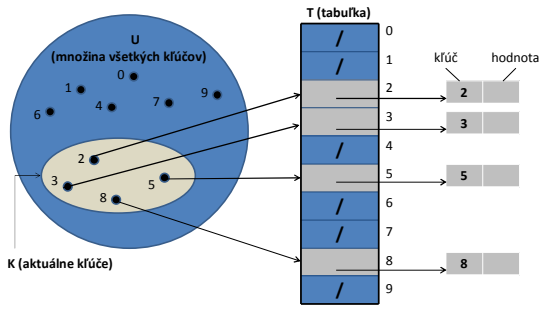
Tabuľka

Tabuľka

- Dátová štruktúra, ktorá asociuje hodnotu s kľúčom.
- V pamäti sa dáta uchovávajú ako dvojica kľúč – hodnota
- K hodnote sa pristupuje pomocou kľúča

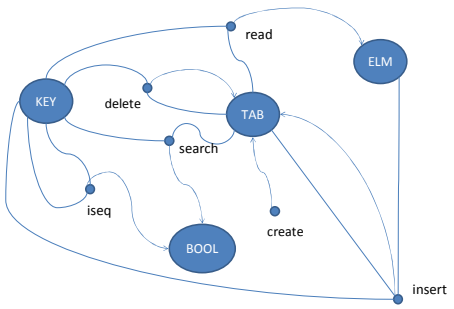


Implementácia dátovej štruktúry tabuľka
- každý kľúč z celej množiny kľúčov U = {0, 1, ..., 9} korešponduje s indexom v tabuľke T
- množina aktuálnych kľúčov K = {2, 3, 5, 8} obsahuje smerník na element (dáta), ostatné sú prázdne resp. sa rovnajú null

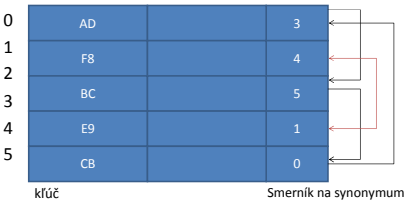
Tabuľka – formálna špecifikácia

- Druhy: TAB, ELM, KEY, BOOL
- Operácie:
  - CREATE() → TAB vytvorenie prázdnej tabuľky
  - INSERT(tab, key, elem) → TAB vloženie prvku
  - READ(tab, key) → ELM výber prvku
  - DELETE(tab, key)→ TAB vymazanie prvku
  - ISEQ(key, key) → BOOL porovnanie 2 prvkov
  - SEARCH(key,tab) → BOOL test, či sa v tabuľke nachádza prvok

Tabuľka



Slovník



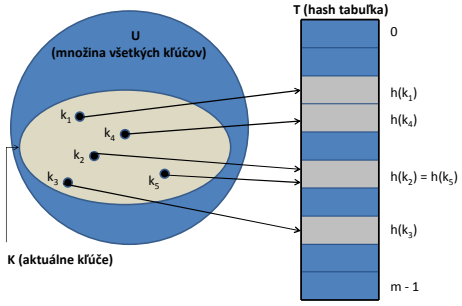
f(AD) = f(BC) = f(CB) f(F8) = f(E9)

Každé slovo obsahuje aj smerník na synonymum. Všetky synonymá potom môžeme získať postupným prechádzaním vzniknutého spájaného zoznamu

Hash tabuľka

- Tabuľka, kde sa kľúč vypočíta pomocou špeciálnej (hešovacej) funkcie
- Príklad jednoduchej hešovacej funkcie:
  - Kľúč = súčet ascii hodnôt jednotlivých znakov
  - Value = OKNO

  OKNO → key = 117 (O) + 113(K) + 116(K) + 117(O)
  = 463



Implementácia dátovej štruktúry hash tabuľka
- použitie hash funkcie h na namapovanie kľúčov k indexom hash tabuľky
- kľúče k_2 a k_5 sú namapované na rovnaký index - kolízia

Hešovacia funkcia

- Základné vlastnosti hešovacej funkcie:

  - Výpočet by nemal byť náročný

  - Mala by byť navrhnutá tak, aby vznikalo čo najmenej kolízií

Aká má byť *h(k)* ?

- Pre tabuľku s *m* položkami:
- ∑ *P(k) = 1/m* pre *j*=0,1,...,*m*-1
  - suma cez *k*: *h(k)=j*
  - ale zvyčajne nepoznáme *P(k)*
- Ak poznáme *P(k)*, napr.:
  - Kľúče sú náhodné reálne čísla nezávislo rovnomerne rozdelené v intervale 0<=*k*<1,
  - *h(k)= floor(k.m)*

Ako navrhnúť *h(k)* ?

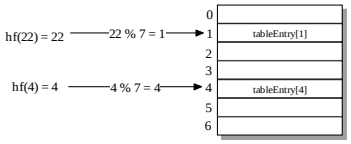
- Pomocou heuristík, opierajúc sa o kvalitatívne znalosti o P.
- Tabuľka symbolov v prekladači:
  - vieme, že často sa vyskytujú v programe symboly, ktoré sa len málo líšia, napr: refA, refB
  - Dobrá h(k) by mala minimalizovať prípady, že takéto symboly pošle na rovnaké miesto v tabuľke.

Návrh *h(k)* metódou delenia

- *h(k)* = *k* mod *m*
  - *m = 16, k = 36, h(36) = 4.*
- *m* nemá byť mocnina 2. Prečo?
  - $m = 2^p$: *h(k)* závisí len od dolných *p* bitov kľúča
- *m* nemá byť mocnina 10. Prečo?
- m má byť prvočíslo nie blízke nejakej mocnine 2.

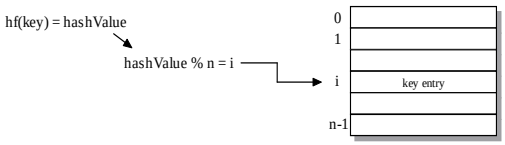
Hešovacia funkcia - príklad

- hf(x) = x, kde x je kladné číslo.
- Tabuľku predstavuje poľe s veľkosťou 7



Hešovanie

```
Hash Value:          hf(key) = hashValue
HashTable index:     hashValue % n
```



Hešovanie

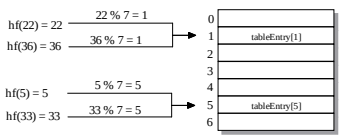
- Hešovacia tabuľka rozdeľuje elementy do série zlinkovaných listov označovaných ako sektory
- Hešovacia funkcia mapuje hodnotu na index v tabuľke. Funkcia poskytuje podobný prístup k elementu ako index v poli.

Hešovanie

- Hešovaciu tabuľku predstavuje pole referencií, kde
  - Hešovacia funkcia má kľúč ako argument a vráti integer hodnotu
  - Výpočtom zvyšku po delení (delí sa veľkosťou tabuľky) získame mapovanie kľúčov na indexy v tabuľke

Hešovacia funkcia - príklad

- Pri zvolenej hešovacej funkcii nastáva kolízia pri každých 2 kľúčoch, ktoré sa navzájom líšia o násobok veľkosti tabuľky
- 36 – 22 = 14 = 2*7 → nastane kolízia



Jednoduchá hešovacia funkcia

- Častokrát je kľúčom reťazec (string)
  - Vo funkcii môžeme kombinovať sekvenciu znakov z reťazca

```
public int hashCode()
{   int hash = 0;

    for (int i = 0; i < n; i++)
        hash = 31*hash + s[i];

    return hash;
}
```

Jednoduchá hešovacia funkcia

Výpočet hašovacej hodnoty pre 3 rôzne reťazce:

Hodnota pre strB je negatívna kôli pretečeniu

```
String strA = "and", strB = "uncharacteristically",
       strC = "algorithm";

hashValue = strA.hashCode();      // hashValue = 96727
hashValue = strB.hashCode();      // hashValue = -2112884372
hashValue = strC.hashCode();      // hashValue = 225490031
```

- ak nastane prípad, že hešovacia funkcia vráti záporné číslo, pribudne problém pri práci s poľom (záporný index neexistuje).

-nasledujúci výpočet však zabezpečí, že index bude vždy kladné číslo:
tableIndex = (hashValue & Integer.MAX_VALUE) % tableSize

Hešovacia funkcia - príklad

- Výpočet hešovacej funkcie pre produkt, ktorý je charakterizovaný sériovým číslom

```
public class Product
{
   private int serialNum;
   public int hashCode()
   {
       long hashValue = serialNum;
       hashValue *= hashValue;
       return (int)(hashValue % Integer.MAX_VALUE);
   }
}
```

Návrh *h(k)* metódou násobenia

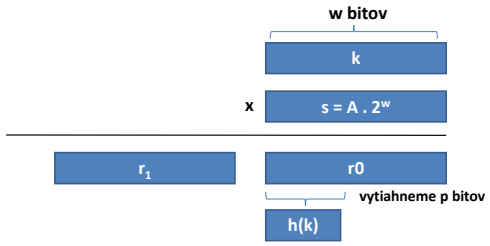
1. Vynásobiť kľúč k zvolenou konštantou *A*, 0<*A*<1 a vybrať zlomkovú časť z *k.A*.
2. Vynásobiť túto hodnotu *m* a vziať celú časť.

h(k) = floor(*m.* (*k.A* mod 1))

kde *k.A* mod 1 označuje zlomkovú časť z *k.A*, t.j. *k.A* – floor(*k.A*)

voľba *A*: podľa Knutha približne sqrt(5) – 1 = 0.6180339887

Hešovacia funkcia - príklad



Násobenie ako metóda hešovania
-w-bitová reprezentácia kľúča sa vynásobí w-bitovou hodnotou s = A.2^w
- p najvyšších bitov nižšej w-bitovej polovice výsledku zvolíme ako hash hodnotu funkcie h(k)

kolízia

- Keď hešovacia hodnota dvoch (alebo viacerých) elementov ukazuje na rovnaké miesto v tabuľke nastáva kolízia. Dva elementy nemôžu byť uložené na rovnakej pozícii v tabuľke.
- Možnosti riešenia problému:
  - Umiestnenie jedného z kolidujúcich elementov na inú pozíciu v tabuľke (*linear probing)*
  - Navrhnutie takej štruktúry, ktorá bude schopná uchovávať viacero elementov s rovnakou hešovacou hodnotou (sekvencia spájaných zoznamov)

Otvorené adresovanie

- Všetky prvky sa ukladajú priamo v tabuľke
- Každá položka tabuľky obsahuje buď prvok reprezentovanej dynamickej množiny (slovníka) alebo NIL.
- Nič sa nezreťazuje, nič nie je mimo tabuľky.
- Miera naplnenia tabuľky je vždy najviac 1.

Vkladanie pri otvorenom adresovaní

- Postupne sa skúša nájsť prázdne miesto
- Nie postupne ďalšieho a ďalšieho suseda *Θ(m)*
- Postupnosť skúšaných miest závisí od kľúča, t.j. hešovacia funkcia dostane ďalší parameter *h(k,0), h(k,1), ..., h(k,m-1)*
- Táto postupnosť m miest/adries musí byť permutáciou
- *0,1,...,m-1*

Hash insert(T, k)

```
insert(T table, k key)
{
       i ← 0

       repeat j ← h(k, i)

              if T[j] = NIL
                     then T[j] ← k
                            return j
              else i ← i + 1

       until i = m

       error “hash table overflow”
}
```

Hash search(T, k)

```
search(T table, k key)
{
       i ← 0

       repeat j ← h(k, i)

              if T[j] = k
                     then return j
              else
                     i ← i + 1

       until T[j] = NIL or i = m

       return NIL
}
```

Lineárne skúšanie (linear probing)

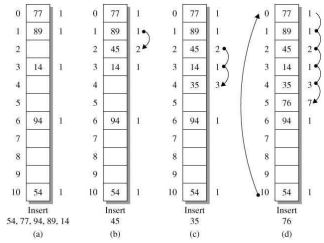
- Vloženie prvku:
  - Na začiatku sa všetky bunky tabuľky označia ako prázdne
  - Aplikuje sa hešovacia funkcia a zvyšok po delení tejto hodnoty veľkosťou tabuľky predstavuje index v tabuľke. Ak je bunka prázdna, vloží sa do nej element
  - Inak sa postupne prehľadávajú ďalšie bunky v poradí a element sa vloží do prvej voľnej.

Lineárne skúšanie (linear probing)

- Uvažujme bežnú hešovaciu funkciu *h'(k): U → {0,1….,m-1}*
- *h(k,i)= (h'(k)+i)* mod *m pre i= 0,1,...,m-1*
- Problém: strapce

Linear Probing

```
tableIndex = x % 11
```



a) vloženie prvkov na ich príslušné pozície bez posunov

b) prvok 45 sme museli posunúť o jedno miesto kvôli obsadenému miestu 1

c) posunutie 35 až o dve miesta (4 namiesto 2)

d) posunutie prvku 76 až na miesto 5 namiesto pôvodného miesta 10

Linear Probing - algoritmus

```
//výpočet indexu tabuľky
int index = (item.hashCode()&Integer.MAX_VALUE)%n

// uloženie pôvodného indexu
int origIndex = index;

//cyklické prehľadávanie tabuľky a hľadanie volnej pozície
// nájde miesto alebo sa tabuľka zaplní(origIndex == index).
do
{
    if table[index] is empty
       insert item in table at table[index] and return
    else if table[index] matches item
       return
    // posunutie v tabuľke
    index = (index+1) % n;
}
while (index != origIndex);
throw new BufferOverflowException();
```

Linear Probing

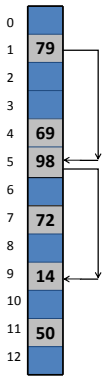
- Táto metóda je vhodná v prípade, ak veľkosť tabuľky je rádovo vačšia ako počet elementov, ktoré sa do nej budú vkladať.
- Dobrá hešovacia funkcia minimalizuje kolízie a ak aj nastanú, tak v tabuľke bude dostatok voľných miest pre vyhľadanie náhradnej pozície

kvadratické skúšanie (quadratic probing)

- Uvažujme bežnú hešovaciu funkciu $h'(k): U \rightarrow \{0,1\ldots,m-1\}$
- $h(k,i)= (h'(k)+c_1.i+c_2.i^2)$ mod m *pre* $i= 0,1,\ldots,m-1$
- Problém: ako zvoliť c_1 a c_2
- Strapce vznikajú sekundárne, menej

dvojité hešovanie

- Uvažujme bežné hešovacie funkcie $h_1(k): U \rightarrow \{0,1\ldots,m-1\}$ $h_2(k): U \rightarrow \{0,1\ldots,m-1\}$
- $h(k,i)= (h_1(k)+i.h_2(k))$ mod m *pre* $i= 0,1,\ldots,m-1$



Hešovacia funkcia - príklad

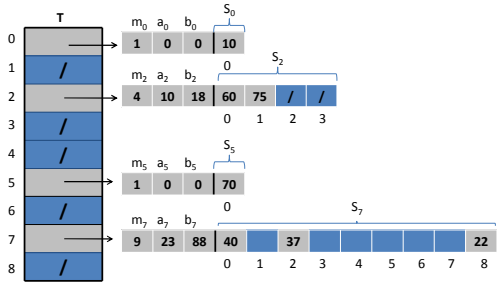
Vkladanie pomocou dvojitého hešovania

- máme hešovaciu tabuľku s veľkosťou 13 s $h_1(k) = k \bmod 13$ a $h_2(k) = 1 + (k \bmod 11)$

- keď $14 \equiv 1 \pmod{13}$ a $14 \equiv 3 \pmod{11}$, tak potom kľúč 14 je vložený na prázdne miesto 9, potom ako sa zistilo, že miesta 1 a 5 sú obsadené

Hešovacia funkcia - príklad

Použitie perfektného hešovania (perfect hashing) na uloženie množiny K = {10, 22, 37, 40, 60, 70, 75}



Hešovacia funkcia - príklad

Použitie perfektného hešovania (perfect hashing) na uloženie množiny K = {10, 22, 37, 40, 60, 70, 75}

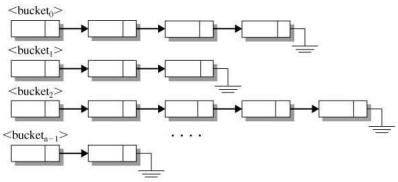
-Základná hešovacia funkcia je *h(k) = ((ak + b) mod p) mod m*, kde *a* = 3, *b* = 42, *p* = 101 a *m* = 9
-Príklad: h(75) = 2, takže objekt 75 sa uloží na miesto s kľúčom 2
-Sekundárna hešovacia tabuľka S_j obsahuje všetky kľúče hešujúce index j
-jej veľkosť je m_j
-a hešovacia funkcia je $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$

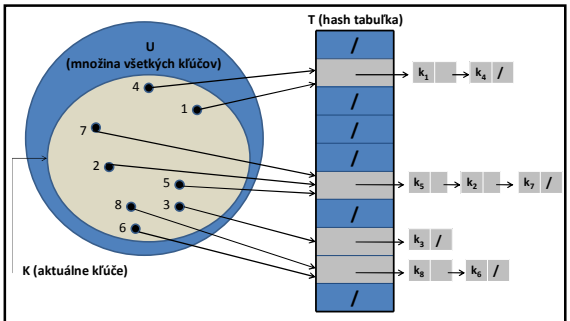
-Keď $h_2(75) = 1$, kľúč 75 je uložený na miesto 1 v sekundárnej hešovacej tabuľke S2

-Takto nie sú žiadne kolízie v sekundárnych hešovacích tabuľkách a vyhľadávanie trvá v najhoršom prípade konštantný čas

Sekvencia spájaných zoznamov

- V tomto prípade definujeme hešovaciu tabuľku ako indexovanú sekvenciu voľne spájaných zoznamov.
- Každý spájaný zoznam sa nazýva bucket a uchováva elementy s rovnakým indexom (rovnakou hešovacou hodnotou)





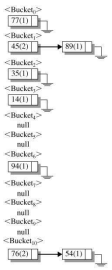
Riešenie kolízií zreťazením
- každý index hash tabuľky T[*i*] obsahuje spájaný zoznam všetkých kľúčov (bucket), ktorých hodnota je *i* napr. $h(k_1) = h(k_4)$ a $h(k_5) = h(k_2) = h(k_7)$

Sekvencia spájaných zoznamov

- Vloženie prvku
  - Hešovacou funkciou sa zistí index bucketu
  - Ak je bucket prázdny vloží sa prvok na prvú pozíciu
  - Ak nie je bucket prázdny, najskôr sa celý prehľadá, či sa v ňom element už nenachádza. Ak nie, tak sa vloží na začiatok.

Sekvencia spájaných zoznamov

Vloženie prvkov: {54, 77, 94, 89, 14, 45, 35, 76}
Veľkosť tabuľky je 11.

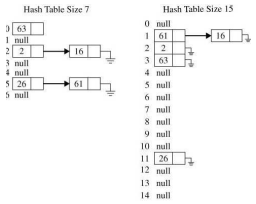


Sekvencia spájaných zoznamov

- V prípade väčšieho počtu kolízií je to výhodnejšia metóda
- Nie je obmedzená pevným počtom prvkov (resp. tak ako je obmedzené pole)

Rehašovanie

- So zvyšujúcim počtom prvkov v hešovacej tabuľke (a zároveň so zvyšujúcim počtom kolízií) sa efektivita vyhľadávania znižuje.
- Rehašovanie predstavuje zväčšenie veľkosti tabuľky, ak súčasná je zaplnená do určitej úrovne.



Hešovacia tabuľka – implementácia

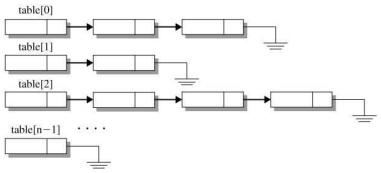
```
Bucket:

private static class Entry<T>
{
   // hodnota v hešovacej tabuľke
   T value;
   int hashValue;
   Entry<T> next;

   Entry(T value, int hashValue, Entry<T> next)
   {
      this.value = value;
      this.hashValue = hashValue;
      this.next = next;
   }
}
```

Hash Class Konštruktor

- Konštruktor vytvorí 12 prázdnych bucketov.



Hash Class

```
public class Hash<T> implements Collection<T>
{
   private Entry[] table;
   private int hashTableSize;
   private final double MAX_LOAD_FACTOR = .75;
   private int tableThreshold;

   private int modCount = 0;

   public Hash()
   {
      table = new Entry[17];
      hashTableSize = 0;
      tableThreshold =
            (int)(table.length * MAX_LOAD_FACTOR);
   }
   . . .
}
```

Hash Class add()

- Pri vložení sa inkrementuje hashTableSize a modCount
- If `hashTableSize ≥ tableThreshold` nastane rehash().
- Veľkosť novej tabuľku bude:

`2*table.length + 1`

Hash add() - 1

```
public boolean add(T item)
{

   int hashValue = item.hashCode() &
         Integer.MAX_VALUE,
         index = hashValue % table.length;
   Entry<T> entry;

  entry = table[index];

// zistenie, či sa hodnota už v zozname nenachádza
  while (entry != null)
   {
      if (entry.value.equals(item))
         return false;

      entry = entry.next;
   }
```

Hash add() - 2

```
modCount++;

   entry = new Entry<T>(item, hashValue,
           (Entry<T>)table[index]);

// vlož hodnotu
   table[index] = entry;
   hashTableSize++;

   if (hashTableSize >= tableThreshold)
      rehash(2*table.length + 1);

   return true;
}
```

Hash rehash() - 1

```
private void rehash(int newTableSize)
{
   Entry[] newTable = new Entry[newTableSize],
           oldTable = table;
   Entry<T> entry, nextEntry;
   int index;


   for (int i=0; i < table.length; i++)
   {

      entry = table[i];
```

Hash rehash() - 2

```
      if (entry != null)
      {

         do
         {
            nextEntry = entry.next;

            index = entry.hashValue % newTableSize;

            entry.next = newTable[index];
            newTable[index] = entry;
```

Hash rehash() - 3

```
            entry = nextEntry;
         } while (entry != null);
      }
   }


   table = newTable;

   tableThreshold =
         (int)(table.length * MAX_LOAD_FACTOR);


   oldTable = null;
 }
```

Hash remove() - 1

```
public boolean remove(Object item)
{

   int index = (item.hashCode() &
         Integer.MAX_VALUE) % table.length;
   Entry<T> curr, prev;

   curr = table[index];
   prev = null;

   while (curr != null)
      if (curr.value.equals(item))
      {
             modCount++;
```

Hash remove() - 2

```
        if (prev != null)
           prev.next = curr.next;
        else
           table[index] = curr.next;


        hashTableSize--;

        return true;
     }
else
     {

        prev = curr;
        curr = curr.next;
     }
return false;
}
```