Hašování (*Vyhledávání metodou transformace klíče*)

Velmi účinnou vyhledávací metodou je hašování. Pro její anglický název *hashing* se poměrně obtížně hledá vhodný překlad do českého jazyka, proto zůstaneme u používání mírně češtině přizpůsobeného anglického názvu.

Datová struktura použitá v hašování pro uložení prvků je tabulka. Tabulka se skládá z řádků. U hašování pro řádky tabulky používáme označení *přihrádky* (v angličtině jsou označovány pojmem *buckets*). V každé přihrádce je místo pro uložení jednoho datového prvku. Počet přihrádek v tabulce, tedy kapacitu tabulky označme *m*. Na jednotlivé přihrádky v tabulce se odkazujeme (adresujeme je) jejich pořadovými čísly 0 až *m*-1. Hašovací tabulka se dá snadno implementovat pomocí pole. Datový typ prvků pole je identický s datovým typem ukládaných údajů a každý prvek pole reprezentuje jednu přihrádku tabulky. Indexování prvků pole odpovídá adresování jednotlivých přihrádek v tabulce.

Základem hašování je hašovací funkce. Je to zobrazení, které hodnotě prvku (nebo vyhledávacímu klíči prvku, pokud prvek je strukturovaný typ) přiřadí číslo některé z přihrádek v tabulce, tedy číslo v rozmezí 0 až *m*-1. Hašovací funkce se typicky skládá ze dvou funkcí. Ta první hodnotu prvku zobrazí na celé (nezáporné) číslo. Druhá celé číslo zobrazí na číslo přihrádky v tabulce, tedy na celé číslo z intervalu <0,*m*-1>. Je zřejmé, že první funkce závisí na datovém typu prvku a sestavujeme si ji sami. Druhá funkce už je víceméně standardní a jen ji použijeme.
Cílem hašovací funkce je rovnoměrné rozmístění prvků v tabulce. Z toho plyne, že první funkce, která převádí hodnotu prvku na celé číslo, by měla mít vlastnosti:

- Zobrazovat hodnoty prvků na co největší počet různých celých čísel.
- Zobrazení na celá čísla by mělo být rovnoměrné (na jednotlivá čísla by se měl zobrazovat přibližně stejný počet prvků, které chceme do hašovací tabulky uložit).

Dalším přirozeným požadavkem na hašovací funkci je, aby její výpočet nebyl příliš časově náročný.

Hašovací funkce pro řetězce

Řetězce jsou častým vyhledávacím klíčem. Řetězec je posloupnost znaků. Jsou různé možnosti, jak znaky zobrazit na celá čísla. Standardně se používá ASCII tabulka nebo kódování Unicode. ASCII tabulka znaky zobrazuje na čísla z intervalu <0,255>. Unicode je nejčastěji zobrazuje na čísla z intervalu <0,65535> (kódování UTF-16).

Řetězec si označme

$$z_1 z_2 \ldots z_k \quad ,$$

kde z_i jsou jednotlivé znaky v řetězci a k je počet těchto znaků v řetězci (délka řetězce).

Jedna z jednodušších funkcí zobrazující řetězec na celé číslo je

$$c_1(z_1 z_2 ... z_k) = p * asc(z_1) + q * asc(z_2) + asc(z_k) + k \quad ,$$

kde p a q jsou zvolené konstanty, nejlépe prvočísla (např. p=127, q=31), protože ty mají nejlepší předpoklady pro rovnoměrné zobrazení do množiny celých čísel. Funkce *asc* převádí znak na jeho ASCII hodnotu (nebo Unicode hodnotu).

Dokonalejší, ale na druhé straně náročnější na výpočet, je funkce

$$c_2(z_1 z_2 ... z_k) = p^{k-1} * asc(z_1) + p^{k-2} * asc(z_2) + p^{k-3} * asc(z_3) + ... + p * asc(z_{k-1}) + asc(z_k) \ ,$$

kde p je konstanta, opět nejlépe prvočíslo (např. p=31). Pro její výpočet je účelné přepsat ji do tvaru

$$c_2(z_1 z_2 ... z_k) =$$
$$p * (... p * (p * (p * asc(z_1) + asc(z_2)) + asc(z_3)) ... + asc(z_{k-1})) + asc(z_k) \ .$$

Druhá část hašovací funkce, která převádí celé číslo na číslo přihrádky v hašovací tabulce, je velmi jednoduchá. Používá se pro ni operace *mod*, což je zbytek po celočíselném dělení. Obecný zápis hašovací funkce je

$$h(x) = c(x) \bmod m \quad ,$$

kde *c(x)* je první část hašovací funkce, která nám převádí hodnotu prvku na celé číslo, m je rozsah (počet přihrádek) hašovací tabulky. Opět je nejlepší zvolit m prvočíslo, protože to nemá žádného netriviálního vlastního dělitele, čímž poskytuje nejlepší předpoklady pro rovnoměrné rozmístění prvků v tabulce.

Vedle prvočíselného počtu přihrádek v tabulce se v praxi používá i počet přihrádek, který je mocninou čísla 2. Tento počet nemá tak dobré předpoklady pro rovnoměrné rozmístění prvků v tabulce, ale výpočet hašovací funkce je snadnější, protože místo operace modulo lze použít jednodušší operaci bitového součinu:

$$h(x) = c(x) \,\&\, (m-1), \quad m = 2^s \quad .$$

Obecně operace dělení a násobení (zejména u malých výpočetních systémů) jsou citelně náročnější než ostatní operace (sčítání, odčítání, bitové operace). V těchto případech se používají hašovací funkce bez násobení a dělení. Jednoduchá hašovací funkce bez náročných operací (násobení, dělení):

$$h(z_1 z_2 ... z_k) = \big((asc(z_1) << 7) - asc(z_1) + (asc(z_2) << 5) - asc(z_2) + asc(z_k) + k\big) \,\&\, (m-1) \ .$$

V ní se využívá vztahů 127=128-1, 31=32-1 a volí se velikost tabulky $m = 2^s$. Uvedená funkce je tímto ekvivalentní s funkcí

$$h(z_1 z_2 ... z_k) = \big(127 * asc(z_1) + 31 * asc(z_2) + asc(z_k) + k\big) \,\&\, (m-1)$$

Hašovací funkce pro datum:

$$c(den, mesic, rok) = p * den + q * mesic + rok \quad ,$$

kde p a q jsou nejlépe prvočísla.

Jiná hašovací funkce pro datum:

$$c(den, mesic, rok) = den \mid mesic << 5 \mid rok << 9 \quad ,$$

kde | označuje bitový součet.

Příklad. Máme navrhnout hašování pro uložení řetězců. Velikost tabulky požadujeme přibližně 500 přihrádek.
Nebližší prvočísla jsou 499 a 503, velikost tabulky zvolíme třeba 503 přihrádek. Pro sestavení hašovací funkce použijeme již uvedenou funkci c_1. Hašovací funkce bude mít tvar:

$$h(z_1 z_2 ... z_k) = (127 * asc(z_1) + 31 * asc(z_2) + asc(z_k) + k) \bmod 503$$

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Příklad. Do tabulky velikosti 11 budeme ukládat řetězce. Hašovací funkci zvolíme

$$h(z_1 z_2 ... z_k) = c(z_1 z_2 ... z_k) \bmod 11 \quad ,$$

kde

$$c(z_1 z_2 ... z_k) = 7 * asc(z_1) + 3 * asc(z_2) + asc(z_k) + k \quad .$$

Do tabulky uložíme jména

h(*Eva*) = (7*69+3*118+97+3) mod 11 = 2
h(*Irena*) = (7*73+3*114+97+5) mod 11 = 9
h(*Pavel*) = (7*80+3*97+108+5) mod 11 = 7
h(*Marta*) = (7*77+3*97+97+5) mod 11 = 8
h(*Ivan*) = (7*73+3*118+110+4) mod 11 = 0
h(*Nina*) = (7*78+3*105+97+4) mod 11 = 5

Číslo přihrádky	Uložené jméno
0	Ivan
1	
2	Eva
3	
4	
5	Nina
6	
7	Pavel
8	Marta
9	Irena
10	

Kdybychom nyní chtěli do tabulky uložit jméno *Helena*, jehož hodnota hašovací funkce je

h(*Helena*) = (7*72+3*101+97+6) mod 11 = 8 ,

zjistíme, že tato pozice už je obsazená jménem *Marta*. Takové kolize, kdy více prvků se zobrazuje na stejnou přihrádku hašovací tabulky, se v hašování vyskytují zcela běžně. V následující části jsou nejpoužívanější způsoby jejich řešení.

Otevřené adresování

Metoda otevřeného adresování v případě, kdy je pozice v tabulce vypočítaná hašovací funkcí obsazena, počítá další pozice tak dlouho, dokud se nenajde volná pozice anebo se nezjistí, že tabulka už je zaplněna.

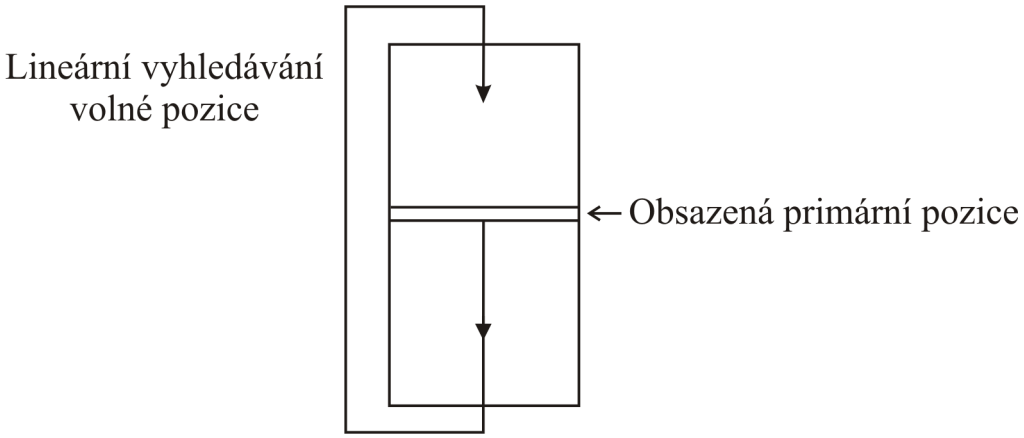
Lineární hledání

Nejjednodušší je lineární hledání, kdy nové pozice počítáme funkcí

$$H(x,i) = (h(x) + i) \bmod m \quad ,$$

kde *h(x)* je výchozí hašovací funkce, *i* je celočíselný parametr a *m* je rozsah tabulky. Je-li tedy primární pozice (*h(x) = H(x,0)*) obsazena, prohledávají se postupně další pozice

$$H(x,1),\ H(x,2),\ldots,\ H(x,m-1)$$



Lineární umísťování za sebou vede k vytváření nežádoucích shluků. Shlukem nazýváme větší počet za sebou následujících obsazených přihrádek tabulky. Pokud je vypočítaná primární pozice obsazená a je přitom uvnitř takového shluku, znamená to při lineárním hledání, že musíme projít všechny přihrádky v tomto shluku za primární pozicí, než se dostaneme k nějaké volné sekundární pozici, abychom prvek do ní mohli uložit. Přitom shluky prodlužují nejen operaci přidání prvku do tabulky, ale také vyhledávání prvku v tabulce. Při vyhledávání začínáme na primární pozici, a pokud na ní prvek není a tato pozice je přitom obsazena (je na ní jiný prvek), procházíme sekundární pozice tak dlouho, dokud prvek nenajdeme nebo se nedostaneme k volné pozici, což je příznakem toho, že hledaný prvek v tabulce není.

Kvadratické hledání

Proto místo lineárního hledání se často používá kvadratické hledání. U něho sice také vznikají shluky, ale už výrazně kratší. Hašovací funkce používaná pro kvadratické hledání má většinou jednoduchý tvar

$$H(x,i) = (h(x) + i^2) \bmod m \ .$$

U ní je ale problém, že během hledání se můžeme touto funkcí dostat znovu na stejnou pozici, kterou jsme již prošli, aniž jsme přitom vyčerpali celou tabulku. Nechť například hodnota hašovací funkce *h* pro nějaký prvek *v* je $h(v)=3$ a mějme tabulku s 5 přihrádkami ($m=5$). Pak u kvadratického hledání dostáváme pozice:

$H(v,0) = (3+0^2) \bmod 5 = 3$
$H(v,1) = (3+1^2) \bmod 5 = 4$
$H(v,2) = (3+2^2) \bmod 5 = 2$
$H(v,3) = (3+3^2) \bmod 5 = 2$

Je zřejmé, že při čtvrtém pokusu (pro i=3) jsme při výpočtu další pozice dostali stejnou pozici, která už byla vypočítána předtím. Ukažme, že to nastane (za předpokladu, že m je prvočíslo), až když prohledáme nejméně polovinu tabulky. V našem příkladu to nastalo, až když jsme prohledali 3 pozice, což je více než polovina z 5. V běžném použití, pokud už tabulka není téměř zaplněna, najdeme nějakou volnou pozici zpravidla dříve, než projdeme polovinu tabulky.

Nechť v kvadratickém hledání dva různé pokusy nalezení nové pozice dají stejný výsledek, tj.

$$H(x, j) = H(x,i), \quad \text{kde} \quad j > i \quad .$$

Dosadíme hašovací funkci

$$(h(x) + j^2) \bmod m = (h(x) + i^2) \bmod m$$

Odstraníme operaci *mod*

$h(x) + j^2 = h(x) + i^2 + K * m$, kde K je nějaké celé číslo.

Odtud

$$j^2 - i^2 = K * m$$
$$(j+i) * (j-i) = K * m \quad .$$

Protože podle předpokladu je $j>i$, je i $K\neq 0$. Dále protože m je prvočíslo, je buďto $j+i$ dělitelné m nebo je $j-i$ je dělitelné m.

Uvažujme, že $j+i$ je dělitelné m, pak

$j + i = L * m$, kde $L\neq 0$ je celé číslo $\Rightarrow$ $j + i \geq m$

$$j + i \geq m \ \wedge \ j > i \ \Rightarrow \ 2 * j + i > m + i \ \Rightarrow \ 2 * j > m \ \Rightarrow \ j > \left\lfloor \frac{m}{2} \right\rfloor \quad .$$

Pokud $j-i$ je dělitelné m, pak

$$j - i = L * m \ \Rightarrow \ j - i \geq m \ \Rightarrow \ j \geq m + i$$
$$j \geq m + i \ \wedge \ i \geq 0 \ \Rightarrow \ j \geq m \quad .$$

Závěr: $j > \left\lfloor \frac{m}{2} \right\rfloor$.

Dvojí hašování

Ještě propracovanější je metoda dvojího hašování. U ní funkce pro hledání má obecný tvar

$$H(x,i) = (h(x) + i * h_2(x)) \bmod m \quad ,$$

kde $h_2(x)$ je další (sekundární) hašovací funkce, která ale nabývá jen hodnoty v rozmezí 1 až m-1 (tedy ne hodnotu 0).

V praxi se sekundární hašovací funkce $h_2(x)$ nejčastěji vytváří využitím primární hašovací funkce $h(x)$, jež je sama sestavena z nějaké výchozí funkce $c(x)$ a má tvar

$$h(x) = c(x) \bmod m \quad .$$

Z ní použijeme její základní část, funkci $c(x)$, a hašovací funkci $h_2(x)$ vytvoříme ve tvaru

$$h_2(x) = 1 + (c(x) \bmod (m-1)) \quad .$$

Funkci

$$H(x,i) = (h(x) + i * h_2(x)) \bmod m$$

upravíme na tvar

$$H(x,i) = (h(x) + (i-1) * h_2(x) + h_2(x)) \bmod m =$$
$$= ((h(x) + (i-1) * h_2(x)) \bmod m + h_2(x)) \bmod m = (H(x,i-1) + h_2(x)) \bmod m \quad .$$

Pozice vložení nyní můžeme počítat rekurzivně

$$H(x,0) = h(x)$$
$$H(x,i) = (H(x,i-1) + h_2(x)) \bmod m \quad \text{pro } i = 1,2,3,\ldots.$$

Příklad. Do tabulky vytvořené v předchozím příkladu chceme uložit další jména

h(*Helena*) = c(*Helena*) mod 11 = 8,
kde c(*Helena*) = 7*72+3*101+97+6 = 910

h(*Bohumil*) = c(*Bohumil*) mod 11 = 8,
kde c(*Bohumil*) = 7*66+3*111+108+7 = 910

h(*Jana*) = c(*Jana*) mod 11 = 8,
kde c(*Jana*) = 7*74+3*97+97+4 = 910

U všech je hodnota hašovací funkce rovna 8, přičemž tato pozice je již obsazena. Zvolíme-li kvadratické hledání, pak další možné pozice jsou

$(8+1^2) \bmod 11 = 9$
$(8+2^2) \bmod 11 = 1$
$(8+3^2) \bmod 11 = 6$
$(8+4^2) \bmod 11 = 2$
$(8+5^2) \bmod 11 = 0$

$(8+6^2) \bmod 11 = 0$
$(8+7^2) \bmod 11 = 2$
$(8+8^2) \bmod 11 = 6$
$(8+9^2) \bmod 11 = 1$
$(8+10^2) \bmod 11 = 9$
$(8+11^2) \bmod 11 = 8$
$(8+12^2) \bmod 11 = 9$
$(8+13^2) \bmod 11 = 1$
$(8+14^2) \bmod 11 = 6$
$(8+15^2) \bmod 11 = 2$
$(8+16^2) \bmod 11 = 0$
$(8+17^2) \bmod 11 = 0$
$(8+18^2) \bmod 11 = 2$
$(8+19^2) \bmod 11 = 6$

Z vypočtených pozic jsou volné pozice 1, 6. Do nich umístíme 2 nová jména:

Číslo přihrádky	Uložené jméno
0	Ivan
1	Helena
2	Eva
3	
4	
5	Nina
6	Bohumil
7	Pavel
8	Marta
9	Irena
10	

V dalším výpočtu pozic se už pozice opakují a pro třetí jméno se volné místo už nenašlo.

Příklad. Do tabulky ukládáme stejná jména jako v předchozím příkladě, pro výpočet pozic použijeme dvojí hašování. Sekundární hašovací funkce bude

$$h_2(x) = 1 + \big(c(x) \bmod 10\big) \quad .$$

Její hodnota pro ukládaná jména je

$h_2(Helena) = h_2(Bohumil) = h_2(Jana) = 1 + 910 \bmod 10 = 1$

Sekundární pozice budou

(8+1) mod 11 = 9
(9+1) mod 11 = 10
(10+1) mod 11 = 0
(0+1) mod 11 = 1
(1+1) mod 11 = 2
(2+1) mod 11 = 3

Z vypočtených pozic jsou volné pozice 10, 1, 3. Do nich umístíme 3 nová jména:

Číslo přihrádky	Uložené jméno
0	Ivan
1	Bohumil
2	Eva
3	Jana
4	
5	Nina
6	
7	Pavel
8	Marta
9	Irena
10	Helena

Vyhledávání v tabulce

Při vyhledávání v hašovací tabulce nejprve vypočítáme hodnotu hašovací funkce pro hledaný prvek *x*. Podíváme se v tabulce do přihrádky, kterou určila hodnota hašovací funkce. Mohou nastat případy:

- Přihrádka je prázdná – hledaný prvek není v tabulce.
- V přihrádce je hledaný prvek *x* – vyhledávání tím úspěšně končí.
- V přihrádce je jiný prvek než *x*. Začneme postupně počítat další možné pozice a srovnávat prvky v nich s hledaným prvkem *x*, dokud buďto hledaný prvek nenajdeme anebo se nedostaneme na prázdnou přihrádku anebo nevyčerpáme všechny možné pozice.

Příklad. Máme vyhledat jméno *Robert* v tabulce z předminulého příkladu.

h(*Robert*)=(7*82+3*111+116+6) mod 11 = 6

Na této pozici je ale jiné jméno - *Bohumil*. Začneme počítat a prohledávat další možné pozice

$(6+1^2) \bmod 11 = 7$ - *Pavel*

$(6+2^2) \bmod 11 = 10$

Vyhledávání skončí na pozici 10, která je prázdná. Jméno *Robert* tedy v tabulce není.

Pseudokódy:

```
SearchLin(T, m, x)    // parametr m je velikost tabulky
  h ← h0 ← hash(x)
  do
    if T[h] = NIL
      return -1
    if T[h] = x
      return h
    h ← (h+1) mod m
  while h ≠ h0
  return -1

SearchQuadr(T, m, x)
  h0 ← hash(x)
  for i ← 0 to m/2
    h ← (h0 + i*i) mod m
    if T[h] = NIL
      return -1
    if T[h] = x
      return h
  return -1

SearchDHash(T, m, x)
  h ← hash(x)
  h2 ← hash2(x)
  for i ← 0 to m-1
    if T[h] = NIL
      return -1
    if T[h] = x
      return h
```

```
    h ← (h + h2) mod m
  return -1

InsertQuadr(T, m, x)
  h0 ← hash(x)
  for i ← 0 to m/2
    h ← (h0 + i*i) mod m
    if T[h] = NIL
      T[h] ← x
      return
  error   // nebyla nalezena volná pozice pro vložení prvku
```

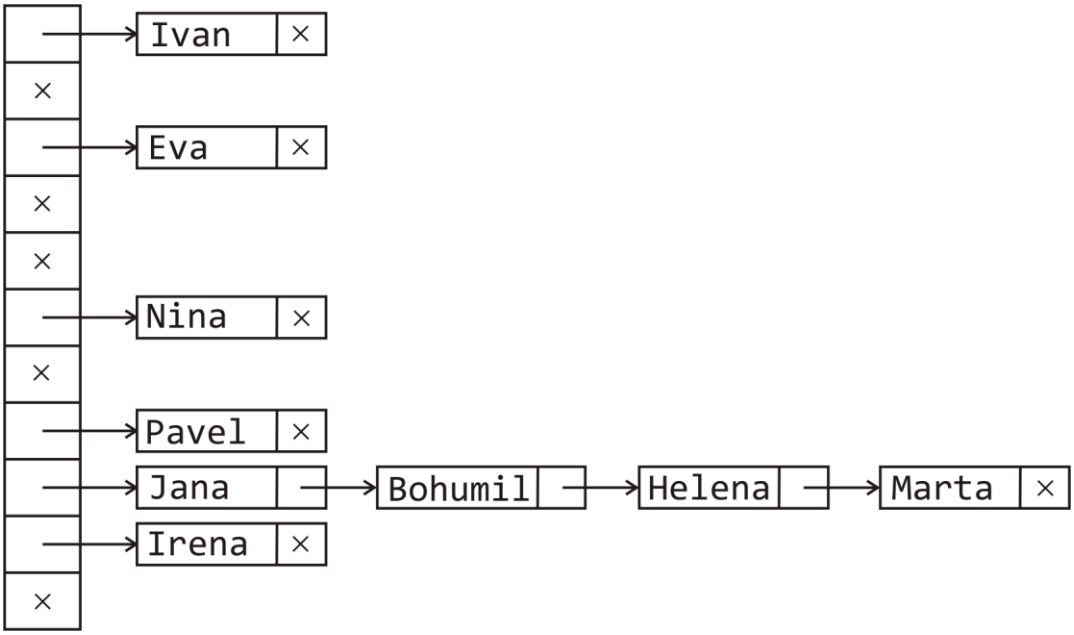
Zřetězení

Předchozí metoda otevřeného adresování má dvě nevýhody:

- Počet prvků, jež lze do tabulky uložit, je omezen její velikostí. Pokud dopředu neznáme, kolik prvků bude do tabulky ukládáno, může se stát, že ji stanovíme malou a dojde k jejímu přeplnění. Následné zvětšení velikosti tabulky může být časově náročně.
- Při vyhledávání, zejména v značně zaplněné tabulce, procházíme v důsledku otevřeného adresování i prvky, které mají jinou hodnotu hašovací funkce, čímž se doba vyhledávání zvyšuje.

Tyto nevýhody odstraňuje metoda zřetězení, která k ukládání dalších prvků se stejnou hodnotou hašovací funkce využívá seznamy. Hašovací tabulka v tomto případě obsahuje ukazatele na začátky (první uzly) jednotlivých seznamů.

Příklad. Tabulka z předchozího příkladu se zřetězením. Všechny prvky jsou nyní uloženy v seznamech.



Pseudokódy:

```
Search(T, x)
  u ← T[hash(x)]
  while u ≠ NIL
    if u.item = x
      return u
    u ← u.link
  return NIL

Insert(T, x)
  u ← new Node
  u.item ← x
  h ← hash(x)
  u.link ← T[h]
  T[h] ← u
```

Odebrání prvku z hašovací tabulky

Pokud potřebujeme do hašovací tabulky prvky nejen přidávat, ale i odebírat, pak je nejvhodnější použít metodu zřetězení pro řešení kolicí. Odebrání se zde udělá vyhledáním prvku v příslušném seznamu a zrušením uzlu, který odebíraný prvek obsahuje.

Pseudokód:

```
Delete(T, x)
  h ← hash(x)
  u ← T[h]
  if u = NIL
    return false
  if u.item = x
    T[h] ← u.link
    delete u
    return true
  while u.link ≠ NIL
    v ← u.link
    if v.item = x
      u.link ← v.link
      delete v
      return true
    u ← u.link
  return false
```

V případě, že odebírání bude v hašovací tabulce s otevřeným adresováním, je realizace odebírání o něco komplikovanější. Budeme rozeznávat tři různé stavy přihrádky tabulky:

- Přihrádka je prázdná – v této přihrádce nikdy nebyl uložen žádný prvek.
- Přihrádka je volná – v této přihrádce byl uložen prvek, ale ten byl pak z tabulky odebrán.
- Přihrádka je obsazena – v této přihrádce je uložen prvek.

Při vyhledávání budeme volné přihrádky přeskakovat. Vyhledávání skončí, až když hledaný prvek najdeme anebo se dostaneme na prázdnou přihrádku (anebo jsme prošli celou tabulku, či její podstatnou část).

Při přidávání budeme hledat první prázdnou nebo volnou přihrádku.

Pseudokódy:

```
SearchQuadr(T, m, x)
  h0 ← hash(x)
  for i ← 0 to m/2
```

```
    h ← (h0 + i*i) mod m
    if T[h] = NIL
      return -1
    if T[h]≠FREE and T[h]=x
      return h
  return -1

InsertQuadr(T, m, x)
  h0 ← hash(x)
  for i ← 0 to m/2
    h ← (h0 + i*i) mod m
    if T[h]=NIL or T[h]=FREE
      T[h] ← x
      return
  error

DeleteQuadr(T, m, x)
  h0 ← hash(x)
  for i ← 0 to m/2
    h ← (h0 + i*i) mod m
    if T[h]=NIL
      return false
    if T[h]≠FREE and T[h]=x
      T[h] ← FREE
      return true
  return false
```

Složitost hašování

Při vyhledávání je složitost dána složitostí výpočtu hašovací funkce a složitostí vlastního vyhledávání. Složitost hašovací funkce závisí na tom, jakým způsobem se počítá. Nicméně ji můžeme považovat za konstantní. Například u řetězců pracujeme buďto s řetězci, které mají omezenou délku, nebo případně můžeme pro výpočet hašovací funkce brát jen omezený počet znaků z řetězce, například uvažujeme jen prvních 20 znaků.

Nejlepší situace v hašování je, když při ukládání prvků (vytváření hašovací tabulky) nedojde k žádné kolizi, pak složitost vyhledávání je viditelně $\Theta(1)$. To ale neposkytuje

žádná doposud popsaná vyhledávací metoda. Běžně kolize nastávají. Složitost přirozeně roste s počtem kolizí v tabulce. Budeme-li uvažovat metodu zřetězení pro řešení konfliktů, pak pro vyhledání prvku potřebujeme počet srovnání, který se pohybuje od 1 (když prvek je okamžitě nalezen) až po maximum z délek seznamů (když prohledáváme v seznamu další prvky se stejnou hodnotou hašovací funkce). Věcně vzato, jestliže zvolíme dostatečně velkou hašovací tabulku vzhledem k předpokládanému množství ukládaných prvků (tj. přiměřeně větší) a zvolíme dostatečně kvalitní hašovací funkci, která rovnoměrně zobrazuje prvky do hašovací tabulky, lze očekávat, že počet srovnání potřebný pro vyhledání prvku bude typicky velmi nízký, tedy jen několik srovnání.
Volba hašovací tabulky je zejména podstatná u metody otevřeného adresování. Volíme ji aspoň o 10% větší, než je očekávaný počet prvků. Empirické testy ukazují, že při 90% zaplnění tabulky je i při nejjednodušší metodě řešení kolizí, kterou je lineární hledání, v průměru zapotřebí 5,5 srovnání pro nalezení libovolného prvku. U sofistikovanějších metod (kvadratické hledání, dvojí hašování) je průměrný počet pokusů ještě menší.
Problémem tabulek s otevřeným adresováním je případ, kdy nelze předvídat počet prvků, které do ní budou uloženy, a nelze tedy vyloučit situaci, že tabulka se zaplní do takové míry, že už se nenajdou volné pozice pro uložení dalších prvků. Pokud by tato situace mohla nastat, je možné tento problém řešit tak, že vytvoříme novou, přiměřeně větší hašovací tabulku (zpravidla její stávající velikost zdvojnásobíme), prvky z dosavadní tabulky do ní přesuneme a předchozí tabulku zrušíme (uvolníme paměť, která pro ni byla vyhrazena). Tato operace je poměrně náročná, protože pro každý prvek před jeho vlastním přesunem do nové tabulky musíme znovu vypočítat hodnotu jeho hašovací funkce a následně najít pro něj volné místo v tabulce.
Pokud shrneme, co jsme o hašovacích tabulkách doposud uvedli, je zřejmé, že hašování je velmi účinná a efektivní metoda vyhledávání a proto jsou hašovací tabulky velmi často používané. Navíc vlastní algoritmus a tím i implementace hašování je poměrně jednoduchý, mnohem jednodušší než třeba vyhledávací stromy.
Použití hašovací tabulky je zejména výhodné v případech, kdy všechny prvky, tedy celou hašovací tabulku lze mít v operační paměti.

Test složitosti vyhledávání při různém zaplnění tabulky

Rozsah následujících testovacích tabulek byl 100 003 přihrádek, do nich byly ukládány náhodně vygenerované řetězce obsahující jen malá písmena (a..z, bez diakritiky), řetězce byly v náhodné délce 5-20 znaků. V tabulkách je uvedeno, kolik bylo zapotřebí průměrně pokusů pro vyhledání prvku při různém zaplnění tabulky.

Hašovací funkce *c2* uvedená v tomto materiálu

% zaplnění	Lineární hledání	Kvadratické hledání	Dvojí hašování	Zřetězení
25	1.17	1.16	1.15	1.12
50	1.50	1.43	1.38	1.25
75	2.48	1.96	1.83	1.37
90	5.61	2.78	2.55	1.45
95	10.75	3.44	3.15	1.47
99	42.09	5.08	4.63	1.49

Jenkinsova hašovací funkce

```
unsigned Hash(const char *Retez)
{ unsigned h=0;
  for (const unsigned char *r = (const unsigned char *)Retez;
                                                  *r!=0; ++r)
  {
    h += *r;
    h += h << 10;
    h ^= h >> 6;
  }
  h += h << 3;
  h ^= h >> 11;
  h += h << 15;
  return h;
}
```

% zaplnění	Lineární hledání	Kvadratické hledání	Dvojí hašování	Zřetězení
25	1.17	1.16	1.15	1.13
50	1.51	1.43	1.39	1.25
75	2.52	1.98	1.86	1.38
90	5.46	2.80	2.55	1.46
95	10.34	3.45	3.15	1.48
99	39.17	5.05	4.62	1.50

CRC 32 (Cyclic redundancy check)

```
static unsigned crc32tab[]={
0x00000000,0x77073096,0xee0e612c,0x990951ba,0x076dc419,0x706af48f,
0xe963a535,0x9e6495a3,0x0edb8832,0x79dcb8a4,0xe0d5e91e,0x97d2d988,
0x09b64c2b,0x7eb17cbd,0xe7b82d07,0x90bf1d91,0x1db71064,0x6ab020f2,
0xf3b97148,0x84be41de,0x1adad47d,0x6ddde4eb,0xf4d4b551,0x83d385c7,
0x136c9856,0x646ba8c0,0xfd62f97a,0x8a65c9ec,0x14015c4f,0x63066cd9,
0xfa0f3d63,0x8d080df5,0x3b6e20c8,0x4c69105e,0xd56041e4,0xa2677172,
0x3c03e4d1,0x4b04d447,0xd20d85fd,0xa50ab56b,0x35b5a8fa,0x42b2986c,
0xdbbbc9d6,0xacbcf940,0x32d86ce3,0x45df5c75,0xdcd60dcf,0xabd13d59,
0x26d930ac,0x51de003a,0xc8d75180,0xbfd06116,0x21b4f4b5,0x56b3c423,
0xcfba9599,0xb8bda50f,0x2802b89e,0x5f058808,0xc60cd9b2,0xb10be924,
0x2f6f7c87,0x58684c11,0xc1611dab,0xb6662d3d,0x76dc4190,0x01db7106,
0x98d220bc,0xefd5102a,0x71b18589,0x06b6b51f,0x9fbfe4a5,0xe8b8d433,
0x7807c9a2,0x0f00f934,0x9609a88e,0xe10e9818,0x7f6a0dbb,0x086d3d2d,
0x91646c97,0xe6635c01,0x6b6b51f4,0x1c6c6162,0x856530d8,0xf262004e,
0x6c0695ed,0x1b01a57b,0x8208f4c1,0xf50fc457,0x65b0d9c6,0x12b7e950,
0x8bbeb8ea,0xfcb9887c,0x62dd1ddf,0x15da2d49,0x8cd37cf3,0xfbd44c65,
0x4db26158,0x3ab551ce,0xa3bc0074,0xd4bb30e2,0x4adfa541,0x3dd895d7,
0xa4d1c46d,0xd3d6f4fb,0x4369e96a,0x346ed9fc,0xad678846,0xda60b8d0,
0x44042d73,0x33031de5,0xaa0a4c5f,0xdd0d7cc9,0x5005713c,0x270241aa,
0xbe0b1010,0xc90c2086,0x5768b525,0x206f85b3,0xb966d409,0xce61e49f,
0x5edef90e,0x29d9c998,0xb0d09822,0xc7d7a8b4,0x59b33d17,0x2eb40d81,
0xb7bd5c3b,0xc0ba6cad,0xedb88320,0x9abfb3b6,0x03b6e20c,0x74b1d29a,
0xead54739,0x9dd277af,0x04db2615,0x73dc1683,0xe3630b12,0x94643b84,
0x0d6d6a3e,0x7a6a5aa8,0xe40ecf0b,0x9309ff9d,0x0a00ae27,0x7d079eb1,
0xf00f9344,0x8708a3d2,0x1e01f268,0x6906c2fe,0xf762575d,0x806567cb,
0x196c3671,0x6e6b06e7,0xfed41b76,0x89d32be0,0x10da7a5a,0x67dd4acc,
0xf9b9df6f,0x8ebeeff9,0x17b7be43,0x60b08ed5,0xd6d6a3e8,0xa1d1937e,
0x38d8c2c4,0x4fdff252,0xd1bb67f1,0xa6bc5767,0x3fb506dd,0x48b2364b,
0xd80d2bda,0xaf0a1b4c,0x36034af6,0x41047a60,0xdf60efc3,0xa867df55,
0x316e8eef,0x4669be79,0xcb61b38c,0xbc66831a,0x256fd2a0,0x5268e236,
0xcc0c7795,0xbb0b4703,0x220216b9,0x5505262f,0xc5ba3bbe,0xb2bd0b28,
0x2bb45a92,0x5cb36a04,0xc2d7ffa7,0xb5d0cf31,0x2cd99e8b,0x5bdeae1d,
0x9b64c2b0,0xec63f226,0x756aa39c,0x026d930a,0x9c0906a9,0xeb0e363f,
0x72076785,0x05005713,0x95bf4a82,0xe2b87a14,0x7bb12bae,0x0cb61b38,
0x92d28e9b,0xe5d5be0d,0x7cdcefb7,0x0bdbdf21,0x86d3d2d4,0xf1d4e242,
0x68ddb3f8,0x1fda836e,0x81be16cd,0xf6b9265b,0x6fb077e1,0x18b74777,
0x88085ae6,0xff0f6a70,0x66063bca,0x11010b5c,0x8f659eff,0xf862ae69,
0x616bffd3,0x166ccf45,0xa00ae278,0xd70dd2ee,0x4e048354,0x3903b3c2,
0xa7672661,0xd06016f7,0x4969474d,0x3e6e77db,0xaed16a4a,0xd9d65adc,
0x40df0b66,0x37d83bf0,0xa9bcae53,0xdebb9ec5,0x47b2cf7f,0x30b5ffe9,
0xbdbdf21c,0xcabac28a,0x53b39330,0x24b4a3a6,0xbad03605,0xcdd70693,
0x54de5729,0x23d967bf,0xb3667a2e,0xc4614ab8,0x5d681b02,0x2a6f2b94,
0xb40bbe37,0xc30c8ea1,0x5a05df1b,0x2d02ef8d };
```

```
unsigned Hash(const char *Retez)
{ unsigned crc=0;
  for (const unsigned char *r = (const unsigned char *)Retez;
                                                *r!=0; ++r)
    crc = crc32tab[(crc ^ *(r++)) & 0xFF] ^ (crc >> 8);
  return crc;
}
```

% zaplnění	Lineární hledání	Kvadratické hledání	Dvojí hašování	Zřetězení
25	1.17	1.17	1.16	1.13
50	1.52	1.45	1.40	1.26
75	2.56	1.99	1.88	1.39
90	5.68	2.83	2.60	1.47
95	10.63	3.55	3.21	1.50
99	62.69	5.30	4.68	1.52

CRC variant

```
unsigned Hash(const char *Retez)
{ unsigned h=0;
  for (const unsigned char *r = (const unsigned char *)Retez;
                                                *r!=0; ++r)
  {
    unsigned h5 = h & 0xF8000000;
    h = h << 5;
    h = h ^ (h5 >> 27);
    h = h ^ *r;
  }
  return h;
}
```

% zaplnění	Lineární hledání	Kvadratické hledání	Dvojí hašování	Zřetězení
25	1.17	1.16	1.15	1.12
50	1.51	1.43	1.38	1.25
75	2.53	1.98	1.84	1.38
90	5.50	2.81	2.54	1.45

95	10.43	3.50	3.14	1.48
99	55.06	5.16	4.58	1.50

Perfektní hašování

Je hašování bez kolizí. Používá se v případech, kdy jsou předem známy všechny prvky, které budou do hašovací tabulky uloženy. Pro tyto prvky se navrhne samostatná (zpravidla velmi specifická) hašovací funkce, při jejímž použití nenastanou žádné kolize.

Minimální perfektní hašování

Je perfektní hašování, kdy navíc po uložení prvků nezůstanou v tabulce žádné volné přihrádky. Počet přihrádek v tabulce je tedy roven počtu uložených prvků. Nalezení takové hašovací funkce je obtížné.

Příklad. Minimální perfektní hašování pro klíčová slova jazyka C.

Hašovací funkce

$$h(z_1 z_2 ... z_k) = kod(z_1) + kod(z_k) + k$$

používá vlastní kódování písmen klíčových slov jazyka C dle následující tabulky:

Písmeno	Kód	Písmeno	Kód	Písmeno	Kód	Písmeno	Kód
a	-5	f	12	l	29	s	7
b	-8	g	-7	m	-4	t	10
c	-7	h	4	n	-1	u	26
d	-10	i	2	o	22	v	19
e	4	k	31	r	5	w	2

Přihrádka	Slovo	Přihrádka	Slovo	Přihrádka	Slovo	Přihrádka	Slovo
0	double	8	const	16	if	24	unsigned
1	case	9	extern	17	switch	25	sizeof
2	char	10	return	18	register	26	long
3	signed	11	while	19	goto	27	float
4	enum	12	else	20	for	28	break
5	continue	13	void	21	auto	29	typedef
6	static	14	do	22	short	30	union
7	default	15	int	23	struct	31	volatile

Další hašovací funkce pro řetězce

```
unsigned RSHash(const char *str)
{
  unsigned b = 378551;
  unsigned a = 63689;
  unsigned hash = 0;
  for (const unsigned char *r = (const unsigned char *)str;
                                                  *r!=0; ++r)
  {
    hash = hash*a + *r;
    a = a*b;
  }
  return hash;
}
```

```
unsigned JSHash(const char *str)
{
  unsigned hash = 1315423911;
  for (const unsigned char *r = (const unsigned char *)str;
                                                  *r!=0; ++r)
  {
    hash ^= (hash<<5) + *r + (hash>>2);
  }
   return hash;
}
```

```
unsigned PJWHash(const char *str)
{
  const unsigned BitsInUnsignedInt = sizeof(unsigned)*8;
  const unsigned ThreeQuarters = (BitsInUnsignedInt*3)/4;
  const unsigned OneEighth = BitsInUnsignedInt/8;
  const unsigned HighBits =
                    0xFFFFFFFF<<(BitsInUnsignedInt-OneEighth);
  unsigned hash = 0;
  unsigned test = 0;
  for (const unsigned char *r = (const unsigned char *)str;
```

```
                                                    *r!=0; ++r)
  {
     hash = (hash << OneEighth) + *r;
     if ((test = hash & HighBits) != 0)
     {
       hash = (hash ^ test>>ThreeQuarters) & ~HighBits;
     }
  }
  return hash;
}
```

```
unsigned ELFHash(const char *str)
{
  unsigned hash = 0;
  unsigned x = 0;
  for (const unsigned char *r = (const unsigned char *)str;
                                                    *r!=0; ++r)
  {
    hash = (hash<<4) + *r;
    if ((x = hash&0xF0000000) != 0) hash ^= x>>24;
    hash &= ~x;
  }
  return hash;
}
```

```
unsigned BKDRHash(const char *str)
{
  unsigned seed = 131; /* 31 131 1313 13131 131313 etc.. */
  unsigned hash = 0;
  for (const unsigned char *r = (const unsigned char *)str;
                                                    *r!=0; ++r)
  {
    hash = hash*seed + *r;
  }
  return hash;
}
```

```
unsigned SDBMHash(const char *str)
{
  unsigned hash = 0;
  for (const unsigned char *r = (const unsigned char *)str;
                                                    *r!=0; ++r)
  {
    hash = *r + (hash<<6) + (hash<<16) - hash;
  }
  return hash;
}
```

```
unsigned DJBHash(const char *str)
{
  unsigned hash = 5381;
  for (const unsigned char *r = (const unsigned char *)str;
                                                    *r!=0; ++r)
  {
    hash = (hash<<5) + hash + *r;
  }
  return hash;
}
```

```
unsigned DEKHash(const char *str)
{
  unsigned hash = len;
  for (const unsigned char *r = (const unsigned char *)str;
                                                    *r!=0; ++r)
  {
    hash = (hash<<5 ^ hash>>27) ^ *r;
  }
  return hash;
}
```

```
unsigned BPHash(const char *str)
{
  unsigned hash = 0;
  for (const unsigned char *r = (const unsigned char *)str;
                                                  *r!=0; ++r)
  {
     hash = hash<<7 ^ *r;
  }
  return hash;
}
```

```
unsigned FNVHash(const char *str)
{
  const unsigned fnv_prime = 0x811C9DC5;
  unsigned hash = 0;
  for (const unsigned char *r = (const unsigned char *)str;
                                                  *r!=0; ++r)
  {
     hash *= fnv_prime;
     hash ^= *r;
  }
  return hash;
}
```

```
unsigned APHash(const char *str)
{
  unsigned hash = 0xAAAAAAAA;
  for (const unsigned char *r = (const unsigned char *)str;
                                                  *r!=0; ++r)
  {
     hash ^= (i&1) == 0 ? (hash<<7) ^ *r * (hash>>3) :
                          ~((hash<<11) + (*r ^ (hash>>5)));
  }
  return hash;
}
```