Hešování

(V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (*hashování*), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín *asociativní pole*.)

Na *heš* se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše {
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

```
void přidej (typ_klíče klíč, typ_hodnoty hodnota) {
  unsigned index = hešovací_funkce (klíč);
  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota) {
  unsigned index = hešovací_funkce (klíč);
  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno || !stejný(klíč, heš[index].hodnota))
      return 0;
  // Našel jsem.
  *hodnota = heš[index].hodnota;
  return 1;
}
```

Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci.* Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota) {
  unsigned index = hešovací_funkce (klíč);
  while (heš[index].obsazeno) {
    index++;
    if (index == K) index = 0;
  }
```

```
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota) {
  unsigned index = hešovací_funkce (klíč);
  while (heš[index].obsazeno) {
    if (stejný (klíč, heš[index].klíč)) {
      *hodnota = heš[index].hodnota;
      return 1;
    }
    // Něco tu je,ale ne to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }
  // Nic tu není.
  return 0;
}
```

Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Hledání může v nejhorším přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $\mathcal{O}(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celou heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $\mathcal{O}(T + H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů, ale je o trošku pracnější), nebo použijeme následující fígl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```
#define mix(a,b,c) {                          \
  a-=b; a-=c; a^=(c>>13);                     \
  b-=c; b-=a; b^=(a<< 8);                     \
  c-=a; c-=b; c^=((b&0xffffffff)>>13);        \
  a-=b; a-=c; a^=((c&0xffffffff)>>12);        \
  b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff; \
  c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff; \
  a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff; \
  b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff; \
  c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff; \
}
```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba:

```
unsigned hash_string (unsigned char *str)
{
  unsigned r = 0;
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce časteji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená méně kolizí a tedy větší rychlost, ale také větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně strávíme přehešováváním konstantní čas na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Poznámky

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé přihrádce (přihrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nová heš bude maximálně $4\times$ větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

Zdeněk Dvořák

Úloha 17-2-1: Prasátko programátorem

Programy pašíka Kvašíka bývají úděsně pomalé a potřebují zrychlit. Lze si je představit jako posloupnosti přiřazení do proměnných, což jsou řetězce znaků složené z malých písmen, velkých písmen a podtržítek. Na pravé straně přiřazení může být buď proměnná nebo operace „+“ nebo „∗“ aplikovaná na dvě proměnné. Tyto operace jsou komutativní, neboli $a + b = b + a$ a $a * b = b * a$.

Vaším úkolem je napsat program, který dostane Kvašíkův program skládající se z N přiřazení a má říci, jak moc ho lze zrychlit, čili říci, kolik nejméně operací „+“ a „∗“ stačí k tomu, aby nový program přiřadil do všech proměnných stejnou hodnotu jako Kvašíkův. Formálně pro každých i prvních řádků Kvašíkova programu musí v novém programu existovat místo, kdy jsou hodnoty všech proměnných z Kvašíkova programu v obou programech shodné. Můžete využívat toho, že operace „+“ a „∗“ jsou komutativní, ale jejich asociativita a distributivita se neberou v úvahu, čili $a + (b + c) \neq (a + b) + c$ a také $(a + b) * c \neq a * c + b * c$.

Příklad: Vlevo je Kvašíkův program, vpravo náš.

```
                        t = b + c;
a = b + c;              a = t;
d = a + b;              d = a + b;
e = c + b;              e = t;
                        s = a * e;
f = a * e;              f = s;
a = d;                  a = d;
g = e * e;              g = s;
```

Zatímco Kvašíkův program potřeboval operací pět, náš si vystačí se třemi, takže výstup programu by měl být „3“.

Úloha 19-4-3: Naskakování na vlak

Naskakování na vlak není věc jednoduchá, při níž se může hodit vědět, jestli se podobný vagón (resp. posloupnost vagónů) vyskytuje ve vlaku víckrát. A tady je příležitost pro vás, abyste se zkoumáním vlaku pomohli.

Vlak si představte jako řetězec délky N, kde každé písmeno představuje jeden vagón (např. `U` je uhelný vagón, `P` je poštovní vůz atp.). Dále máte dáno číslo k ($k \leq N$) a máte zjistit, kolik navzájem různých podřetězců délky k se v řetězci (tedy ve vlaku) vyskytuje. Zároveň tyto podřetězce a počty jejich výskytů vypište. Pozor, vlak už se blíží, takže byste to měli spočítat pekelně rychle.

Příklad: Pro řetězec (vlak) `UPDUPDUDUP` a $k = 3$ jsou nalezené podřetězce

`UPD`	$2\times$
`PDU`	$2\times$
`DUP`	$2\times$
`DUD`	$1\times$
`UDU`	$1\times$