Rozhledy matematicko-fyzikální

Zdeněk Dvořák; Martin Mareš; Milan Straka

Recepty z programátorské kuchařky Korespondenčního semináře z programování, VI. část

Rozhledy matematicko-fyzikální, Vol. 83 (2008), No. 1, 24–30

Persistent URL: http://dml.cz/dmlcz/146092

Recepty z programátorské kuchařky Korespondenčního semináře z programování, VI. část*)

Zdeněk Dvořák, Martin Mareš, Milan Straka, MFF UK Praha

V šestém dílu programátorské kuchařky si povíme něco o hešování[1]. 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 mohou být konstanty, kterými si příslušné příkazy reprezentujeme.
- 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.

*) Informace o Korespondečním semináři z programování pořádaného Matematicko-fyzikální fakultou Univerzity Karlovy v Praze lze nalézt na webové stránce `http://ksp.mff.cuni.cz`.

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

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 x, $0 \leq x < K$, 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 rovno $1.3N$ až $2N$). 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 rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být přibližně $1/K$.

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:

```
type položka_heše = record
  obsazeno : boolean;
  klíč : typ_klíče;
  hodnota : typ_hodnoty;
  end;
var heš : array [0..K - 1] of položka_heše;
```

Prvek `klíč` bude v tomto poli uložen na pozici `hešovací_funkce (klíč)`. Operace naprogramujeme zřejmým způsobem:

```
procedure přidej (klíč : typ_klíče;
             hodnota : typ_hodnoty);
  var index : integer;
begin
  index := hešovací_funkce (klíč);
  { Kolize nejsou, čili heš[index].obsazeno = false. }
  heš[index].obsazeno := true;
  heš[index].klíč := klíč;
  heš[index].hodnota := hodnota;
```

```
end;
function najdi (klíč : typ_klíče; var hodnota : typ_hodnoty)
: boolean;
  var index : integer;
begin
  index := hešovací_funkce (klíč);
  { Nic tu není nebo je tu něco jiného. }
  if not heš[index].obsazeno or
    not stejný(klíč, heš[index].klíč) then
    najdi := false
  else
    begin
      { Našel jsem. }
      hodnota := heš[index].hodnota;
      najdi := true;
    end;
end;
```

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 – musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci*, protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se mohou spojit. Implementace pak vypadá takto:

```
procedure přidej (klíč : typ_klíče;
             hodnota : typ_hodnoty);
  var index : integer;
begin
  index := hešovací_funkce (klíč);
```

```
  while heš[index].obsazeno do
    begin
      inc (index);
      if index = K then index := 0;
    end;
  heš[index].obsazeno := true;
  heš[index].klíč := klíč;
  heš[index].hodnota := hodnota;
end;
function najdi (klíč : typ_klíče; var hodnota : typ_hodnoty)
: boolean;
  var index : integer;
begin
  index := hešovací_funkce (klíč);
  while heš[index].obsazeno do
    begin
      if stejný (klíč, heš[index].klíč) then
        begin
          hodnota := heš[index].hodnota;
          najdi := true;
          exit;
        end;
      { Něco tu je, ale ne to, co hledám. }
      inc (index);
      if index = K then index := 0;
    end;
  { Nic tu není. }
  najdi := false;
end;
```

Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce a hledání bude přeskakovat postupně všechny, čili složitost může být až $O(NT + H)$, 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 celý heš (v lineárním čase).

Nicméně to se obvykle nestane – pokud velikost pole bude dost velká a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hle-

dání i přidávání bude jen $O(T+H)$. 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, že nelze prostě nastavit u mazaného prvku „obsazeno“ na `false`. 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 trik: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme 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 „slijí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu:

```
procedure mix(var a, b, c : integer)
begin
  a := a - b; a := a - c; a := a xor (c shr 13);
  b := b - c; b := b - a; b := b xor (a shl 8);
  c := c - a; c := c - b; c := c xor (b shr 13);
  a := a - b; a := a - c; a := a xor (c shr 12);
  b := b - c; b := b - a; b := b xor (a shl 16);
  c := c - a; c := c - b; c := c xor (b shr 5);
  a := a - b; a := a - c; a := a xor (c shr 3);
  b := b - c; b := b - a; b := b xor (a shl 10);
  c := c - a; c := c - b; c := c xor (b shr 15);
end;
```

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 P a za hodnotu funkce vzít zbytek klíče po dělení P. S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu. Proč právě prvočísla? Kdybychom například zvolili P sudé, hešovací funkce by přiřazovala sudým číslům sudé hodnoty a lichým číslům liché. V případě, že by všechny klíče byly sudé, by se tedy využívala pouze polovina hešovací

tabulky. Podobně je dobré se vyhnout i dalším malým dělitelům.

Rozumná funkce pro hešování řetězců je třeba:

```
function hešuj_řetězec (s : string) : integer;
  var r, c, i : integer;
begin
  r := 0;
  for i := 1 to length (s) do
    begin
      c := ord (s[i]);
      r := r * 67 + c - 113;
    end;
  hešuj_řetězec := r;
end;
```

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 častěji 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 možná větší rychlost, ale také větší plýtvání pamětí), 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 z N prvků na $2N$ prvků přibýt alespoň N prvků, čili průměrně provádíme jedno přehešování 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 při nafukování můžeme takové prvky opravdu smazat a konečně je odečíst z počtu obsazených prvků.

Pár poznámek na závěr:

- 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 nový heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.
- Je zajímavé heš srovnat s binárními vyhledávacími stromy, které jsme si ukázali ve čtvrtém dílu kuchařky [1]. Ve své základní verzi mohou mít operace se stromy až lineární časovou složitost, nicméně jsou-li vkládané hodnoty dostatečně náhodné, složitost každé operace je $O(\log N)$. Navíc, použijeme-li vyvažované stromy, lze časovou složitost $O(\log N)$ zaručit. Hešovací tabulka dosahuje pro náhodné vstupy konstantní časové složitosti, nicméně v obecnosti neexistuje žádná modifikace, která by tuto složitost zaručila. V praxi bývá hešování rychlejší než vyhledávací stromy, a také se snáze implementuje. Stromy se proto typicky používají, jestliže potřebujeme zaručené omezení na časovou složitost. Občas se také hodí, že stromy podporují oproti heším některé další operace, například nalezení minima.

Literatura

[1] Kráľ, D., Mareš, M., Valla, T.: Recepty z programátorské kuchařky Korespondenčního semináře z programování – IV. část. *Rozhledy matematicko-fyzikální* **82**, 1 (2007), s. 22–35.