Hash funkcije - pripremni materijal

SPA2, DMI-PMF-UNS, www.dmi.uns.ac.rs

2019

Contents

Heš kod i jednakost u programskom jeziku Java

U programskom jeziku Java u svakoj klasi mogu (i često i trebaju) da postoje metodi

```
public int hashCode()
public boolean equals(Object o)
```

Oni omogućavaju vrlo efikasne operacije nad velikim skupovima objekata, ali je bitno da uzajamno imaju konzistentno ponašanje.

Tačnije, ako su dva objekta jednaka u skladu sa `equals` tada im i heš vrednost koju vraća `hashCode` mora biti ista. Obrnutno ne mora da važi, odnosno može postojati neograničeni broj različitih objekata koji imaju isti heš kod. Isto tako ako dva objekta imaju različih heš, tada oni ne bi smeli biti jednaki kad se pita `equals`. No i ovde ne važi obrnuto, odnosno ako su dva objekta različiti, heš kod im ne mora biti različit.

Zbog ovih podrazumevanih odnosa je jako bitno da se ova dva metoda pišu zajedno, odnosno da se ne menja samo jedan bez promena u drugom.

U opštem slučaju ako se ovo ponašanje ne promeni reimplementacijom ovih metoda, objekti u Javi će za ova dva metoda da vraćaju vrednosti koje se odnose na sam pokazivač u memoriji. Odnosno dva objekta su isti samo ako se bukvalno isti objekat u memoriji, bez obzira na to da li imaju isti sadržaj. Isto tako će sam heš kod biti generisan od vrednosti samog pokazivača, a ne od sadržaja objekta. Jako često ovo nije poželjno ponašanje, te se zbog toga moraju pisati svoji metodi.

Heš kod je uvek poželjno pisati tako da se dobija što više različitih vrednosti za različite instance objekata. U idealnom slučaju bi se za svaki različit objekat dobijao različit heš kod, međutim to u praksi često nije moguće.

Klasa `String` ima redefinisana oba ova metoda. Heš kod koji se dobija za ovu klasu je veoma kvalitetan i uglavnom se vredi osloniti na njega za polja ovog tipa.

Nažalost, nizovi nemaju nijednu od ove dve funkcije definisane kako treba, pa kada se radi sa njima, mora se uložiti napor da se pojedinačni elementi u nizu porede, odnosno hešuju.

Ocene kvaliteta funkcije

Kvalitetna heš funkcija ravnomerno raspoređuje elemente po celoj tabeli, odnosno dobijamo podskupove sličnih veličina.

Za poređenje kvaliteta heš funkcija se mogu koristi različite mere. Za ove potrebe je napravljena klasa `StatSet` koja implementira klasični Java skup (zapravo u pozadini i koristi `HashSet` za skladištenje podataka), ali osim toga vodi računa i o tome kako su se elementi rasporedili i može da da informacije o tome. Najlakše se dolazi do većine ovih informacija metodom `printStats`.

Prva mera kvaliteta je broj različih heš vrednosti elemenata u skupu, a ispisuje se zajedno sa procentom u odnosu na ukupni broj elemenata u skupu. U idealnom slučaju je ovo 100%, odnosno svi elementi imaju različite vrednosti, mada nekad u realnim primenama ovo nije lako izvodljivo.

Sledeći bitan uvid u raspodelu elemenata je koliko su veliki podskupovi, odnosno koliko su veliki lanci pretraga pri operaciama nad skupom. Ispisuje se prosečna vrednost dužine lanaca, ali i standarno odstupanje (npr 2.12 +- 1.11). U idealnoj raspodeli svi lanci su dužine tačno 1.

Osim ovog podatka se ispisuje i dužina najdužeg lanca u skupu, koja daje predstavu o tome koliko je pretraživanje u najgorem slučaju. Ovaj broj takođe treba da je što manji.

Program ispisuje i vrednosti chi^2 testa, o čemu postoji još komentara na kraju ovog dokumenta, ali se inače neće dublje razmatrati, nego se ostavlja zainteresovanima da pogledaju.

Primeri

U sledećim primerima će se razmatrati različiti tipovi podataka i kako se oni efikasno mogu skladištiti u heš tabelama. Podaci se smeštaju u instancu `StatSet` koja nam daje statistike o njihovoj raspodeli. Svi dati primeri se mogu pokretati iz odgovarajućih klasa pošto postoje `main` metodi koji pokreću program `TestHash` sa odgovarajućim parametrima.

Korišćenje kancelarija

Treba da čuvamo podatke o korišćenju kancelarija u toku jednog dana i da omogućimo što bržu proveru da li je neka konkretna osoba koristila neku konkretnu kancelariju. Podaci vezuju prezimena za identifikacione brojeve soba. Naravno moguće je da više ljudi koristilo istu kancelariju, a moguće je i da je neko više puta koristio istu – ovo će se u tabeli skladištiti samo jedan put, pošto nas samo zanima ko je bio unutra, a ne koliko puta.

Uzmimo sledeću definiciju klase

```
class Kancelarija {
   private int broj;
   private String prezime;
   ....
}
```

U datim fajlovima nema više od~100 prostorija, a i postoji samo oko~60 različitih prezimena (tj zaposlenih).

Broj sobe tipa `int` i deluje kao "očigledan" kandidat za heš kod. Validno bi bilo da se ova informacija vraća kao heš kod:

```
public int hashCode() {
    return broj;
}
```

Pogledajmo primere učitavanja većeg broja podataka u heš tabele sa različitim brojem podskupova, ako imamo tako definisanu heš funkciju:

```
Number of elements: 4885
Different values:   100 ( 2.05 %)
Avg. search chain len.: 48.85 +- 2.99
Longest search chain:   56
```

Budući da ima samo 100 soba, može biti i samo 100 različitih heš vrednosti, te rezultati nisu zadovoljavajući.

Slično važi i za prezimena, budući da ih ima samo šestdesetak, bez obzira na koliko je dobar heš kod u klasi `String`, možemo imati samo šestdesetak različitih rezultata, odnosno dobijamo još gore rezultate. Bitna razlika u odnosu na prethodnu verziju gde smo koristili samo `int` koji je primitivni tip, ovde se mora

voditi računa i o tome da samo polje može biti `null`, tj ne sme se desiti da `hashCode` baci izuzetak zbog ovoga.

```
    public int hashCode() {
        if (prezime != null) {
        return prezime.hashCode();
    } else {
        return 0;
    }
    }
```

```
Number of elements: 4885
Different values:   61 ( 1.25 %)
Avg. search chain len.: 80.08 +- 3.96
Longest search chain:   89
```

Očigledno je potrebno kombinovati ove dve vrednosti, odnosno iskoristiti sve različitosti u podacima koje postoje.. Možemo ih jednostavno samo sabrati:

```
    public int hashCode() {
        int rez = 0;
        if (prezime != null) {
            rez += prezime.hashCode();
        }
        rez += broj;
        return rez;
    }
```

Dobijamo izuzetne rezultate u testovima, pošto su sve vrednosti različite:

```
Number of elements: 4885
Different values:     4885  (100.00 %)
Avg. search chain len.:  1.00 +-  0.00
Longest search chain:       1
```

Istovremeno je bitno da se i `equals` definiše da koristi ista polja. Metod zahteva da parametar bude tipa `Object`, pa uvek moramo da krenemo od toga. Najjednostavnija varijanta zahteva da proverimo da li je prosledjeno nešto što je iste klase kao i naš objekat, pa tek onda možemo da menjamo tip i da poredimo polja.

```
    public boolean equals(Object o) {
        // proveravamo da li je objekat sa
        // kojim se poredimo Kancelarija
        if (o instanceof Kancelarija) {
            // pretvaramo objekat u kancelariju
            Kancelarija k2 = (Kancelarija) o;
            // poredimo polja
            if (prezime.equals(k2.prezime) && broj == k2.broj) {
                return true;
```

```
            }
        }
        return false;
    }
```

Međutim potpuna verzija `equals` metoda treba da proveri da li je u pitanju baš ista klasa kao naša (moguće da je neki naslednik koji tehnički ne mora biti isti), te da tu uzme u obzir i da je moguće da nam je prosleđen `null` što nikako ne može biti isto kao i naš objekat. Dalje se može ubrzati postupak proverom da li su u pitanju pokazivači na isto mesto pošto nema potrebe za proverama onda. Takođe se preporučuje da se polja porede jedno po jedno, pošto je kod pregledniji i lakše se menja, nego kad se kombinuju upiti kao gore.

Potpuna verzija bi izgledala ovako:

```
    public boolean equals(Object o) {
        // Objekat je identican
        if (this == o) {
            return true;
        }
        // Null je uvek razlicit
        if (o == null) {
            return false;
        }
        // Ako su klase razlicite, objekti ne mogu bili jednaki
        if (getClass() != o.getClass()) {
            return false;
        }

        // pretvaramo objekat u kancelariju
        Kancelarija k2 = (Kancelarija) o;

        // Prvo proveravamo broj
        if (broj != k2.broj) {
            return false;
        }

        // A potom prezime
        if (!Objects.equals(prezime, k2.prezime)) {
            return false;
        }

        // Proverili smo polja i sva su jednaka
        return true;
    }
```

Razmotrimo sledeću modifikaciju klase: recimo da se i dalje želi samo pratiti u tabeli ko je bio u kojoj kancelariji, ali da su negde već definisani objekti u

kojima se skladišti i vreme poslednjeg pristupanja, ili recimo neki dodatni opis zašto je neko bio u kancelariji.

```
class KancelarijaVreme {
   private int broj;
   private String prezime;

   private int pristupDan, pristupSat;

   private String komentar;
}
```

Budući da za naše potrebe nije bitno kad je pristupano niti zbog čega, i dalje bi trebali da koristimo iste kodove za `equals` i za `hashCode` pošto nam je dosta da ubacimo samo jedan objekat za svako pojedinačno korišćenje. Negde druge se naravno oni mogu koristiti za detaljije zapise, ali nas to trenutno ne interesuje.

Gadjanje mete

Potrebno je da čuvamo podatke o rezultatima gadjanja mete. U nizu se nalazi do 20 brojeva izmedju i uključujući 0 i 10.

Recimo da je klasa definisana na sledeći način:

```
public class Gadjanje {
    private int[] rezultati;
    private static int MAX_DUZ = 20;
    ...
}
```

Pošto je moguće (i vrlo verovatno zapravo) da neko nije gađao svih 20 puta, u podacima će biti nizovi različitih dužina. U skladu sa tim je i format za unos takav da je prvi broj u redu zapravo broj gađanja, koji određuje veličinu niza.

Na početku je već bilo napomenuto da nizovi nemaju `hashCode` i `equals` implementirane na načine koji bi odgovarao ovoj situaciji, tako da se mora implentirati pojedinačno poređenje elemenata, naročito kod ovakvih situacija sa dodatnim značenjima elemenata.

Funkcija jednakosti bi izgledala ovako:

```
public boolean equals(Object o) {
    // Objekat je identican
    if (this == o) {
        return true;
    }
    // Null je uvek razlicit
    if (o == null) {
        return false;
    }
```

```
        // Ako su klase razlicite, objekti ne mogu bili jednaki
        if (getClass() != o.getClass()) {
            return false;
        }

        Gadjanje o2 = (Gadjanje) o;

        // proveravamo da li su polja null pre dalje provere
        if (rezultati == null && o2.rezultati != null) {
            return false;
        }
        if (rezultati != null && o2.rezultati == null) {
            return false;
        }

        // ako u obe instance nije null, poredimo delove
        if (rezultati != null && o2.rezultati != null) {
            // proverimo duzinu.
            if (o2.rezultati.length != rezultati.length) {
                return false;
            }
            // ako je ista duzina proveravamo elemente
            for (int i = 0; i < rezultati.length; i++) {
                if (o2.rezultati[i] != rezultati[i]) {
                    // cim je nesto razlicito nisu isti
                    return false;
                }
            }
        }
        // ako nije bilo razlika, vracamo da je sve ok
        return true;
    }
```

"Očigledno" rešenje je samo sabrati sve ove vrednosti i vratiti broj. Međutim ovakvo rešenje u najboljem slučaju daje maksimalno 200 različitih vrednosti, pri čemu bi se očekivala i dosta neravnomerna raspodela (sume pri sredini su daleko verovatnije od najviših i najnižih). Na konkretnim podacima su vrednosti u tabeli nešto gore od očekivanih 200 vrednosti:

```
Number of elements: 9049
Different values:   139 ( 1.54 %)
Avg. search chain len.: 65.10 +- 37.98
Longest search chain:   118
```

U ovakvoj strukturi, pozicija u nizu bi trebalo da bude značajna za krajnje heš vrednosti. Da bi se to postiglo jedan pristup je množenje brojeva sa svojim pozicijama ili različitim (prostim) brojevima.

Za ovaj primer u kome je dužina niza raznolika i ima više "početnih" elemenata je dodatno pogodno da se težine primenjuju unazad, pa recimo možemo uvesti da prvi element ima težinu 21, a poslednji u nizu od 20 elemenata težinu 2.

```
    public int hashCode() {
        int rez = 0;
        if (rezultati != null) {
            for (int i = 0; i < rezultati.length; i++) {
                rez += rezultati[i] * (MAX_DUZ + 1 - i);
            }
        }
        return rez;
    }
```

Vrednosti su dosta bolje, iako još uvek nisu na viskom nivou.

```
Number of elements: 9049
Different values:   1411 (15.59 %)
Avg. search chain len.: 6.41 +- 3.89
Longest search chain:   20
```

Problem je zapravo što brojevi nisu dovoljno različiti. Pošto znamo da su rezultati najviše 10, možemo to iskoristiti da u suštini sastavimo višecifreni broj na sledeći način:

```
for (int i = 0; i < rezultati.length; i++) {
    rez = (rez + rezultati[i]) * 10 ;
}
```

Ovim smo dobili dosta bolje vrednosti:

```
Number of elements: 9049
Different values:   8911 (98.47 %)
Avg. search chain len.: 1.02 +- 0.14
Longest search chain:   4
```

Možemo uočiti i sledeće - ako na početku nekog niza stoje 0, one će biti ignorisane u ukupnom zbiru, te samim tim možemo imati različite nizove koji vraćaju iste vrednosti. Ovo možemo ispraviti tako što ćemo pojedinačne rezultate povećavati za 1, odnosno pomeriti ih iz opsega 0-10 na 1-11. U skladu sa tim treba da povećamo i broj kojim množimo na 11.

```
public int hashCode() {
    int rez = 0;
    if (rezultati != null) {
        for (int i = 0; i < rezultati.length; i++) {
            rez = (rez + rezultati[i] + 1) * 11 ;
        }
    }
```

```
    return rez;
}
```

```
Number of elements: 9049
Different values:   9049 (100.00 %)
Avg. search chain len.: 1.00 +- 0.00
Longest search chain:   1
```

Tabla za igru XO

Zadatak nam je da efikasno čuvamo podatke o pozicijama u igri XO, odnosno matrice 3x3, u kojima su vrednosti -1 za "X", 0 za prazno polje i 1 za "O".

Npr:

```
 1 0 -1  O _ X
 1 1 -1  O O X
-1 0  1  X _ O
```

predstavlja poziciju u kojoj je "O" pobedio/la.

Polja klase bi se mogla definisati na sledeći način:

```
public class XO  {
    public static final int DIM = 3;
    private final int[][] tabla = new int[DIM][DIM];
....
}
```

Poredjenje ovakvih objekata bi naravno trebalo da pojedinačno ispituje elemente matrice, na primer:

```
    public boolean equals(Object o) {
        // Objekat je identican
        if (this == o) {
            return true;
        }
        // Null je uvek razlicit
        if (o == null) {
            return false;
        }
        // Ako su klase razlicite, objekti ne mogu bili jednaki
        if (getClass() != o.getClass()) {
            return false;
        }

        // menjamo tip da mozemo da poredimo
        XO o2 = (XO) o;
        // posto je u ovoj klasi uvek inicijalizovano polje table
        // i uvek je DIM x DIM ne moramo proveravati null
```

```
        for (int i = 0; i < DIM; i++) {
            for (int j = 0; j < DIM; j++) {
                if (o2.tabla[i][j] != tabla[i][j]) {
                    return false;
                }
            }
        }
        return true;
    }
```

Za ovaj primer vredi prvo razmotriti da neke relativno očigledne tehnike nisu efikasne: sumiranje ovakve matrice daje vrednosti do plus ili minus 9, odnosno ima samo 19 različitih vrednosti, što bi predstavljalo jako lošu pokrivenost tabele. Dodatno, vrlo često će rezultat biti 0 i uopšteno će biti više tablica koje su bliže "sredini" rezultatom, što opet dovodi do loše ravnomernosti raspoređivanja elemenata.

Dakle ovako definisan metod za heš će davati dosta loše rezultate:

```
    public int hashCode() {
        int rez = 0;
        for (int i = 0; i < DIM; i++) {
            for (int j = 0; j < DIM; j++) {
                rez += tabla[i][j];
            }
        }
        return rez;
    }
```

```
Number of elements: 4424
Different values:   18 ( 0.41 %)
Avg. search chain len.: 245.78 +- 253.74
Longest search chain:   674
```

Efikasnost se u ovakvoj strukturi dobija davanjem "težine" elementima. Odnosno ne treba gledati jednako ako je "1" u gornjem levom uglu i u sredini. Npr u jednostavnijim varijantama se može zbir iz svakog reda množiti različitim brojem. Još bolji rezultati se dobijaju uvodjenjem još detaljnijih "težina": pri sumiranju reda množiti različito, ili čak imati specijalne proste brojeve za svako polje ili uvoditi dodatne operacije.

Takodje treba uzeti u obzir i to da sabiranje pozitivnih i negativnih vrednosti može da dovede do "međusobnog poništavanja", te da nula vrednosti ne pomažu zbiru, naročito kad se uvedu težine koje se množe sa vrednostima polja. Jedan način da se ovo uradi u konkretnom slučaju je da se pri računanju heša vrednosti u matrici sabiraju sa 2, pa time imamo 1,2,3 umesto -1,0,1, što dovodi do veće različitosti.

Jedno od mogućih rešenja koje daje dobre rezultate je zapravo jako slično onome

viđeno kod niza - ako stalno množimo sa brojem koji je veći od mogućih vrednosti jednog elementa, automatski su sva mesta pomnožena različitim koeficijentom. Ovo možemo kombinovati sa pomeranjem pojedinačnih elemenata na pozitivne vrednosti, mada se dobri rezultati mogu postići i bez toga.

```
public int hashCode() {
    int rez = 0;
    int koef = 3;
    for (int i = 0; i < DIM; i++) {
        for (int j = 0; j < DIM; j++) {
            rez = koef * (rez + (tabla[i][j] + 2));
        }
    }
    return rez;
}
```

```
Number of elements:   4424
Different values:     4424  (100.00 %)
Avg. search chain len.:  1.00 +-  0.00
Longest search chain:       1
```

Dodatne napomene

Hi kvadrat ocena kvaliteta heš funkcije

Nešto komplikovanija mera je χ^2 *(hi-kvadrat)* test.

$$\chi^2 = \sum_{i=1}^{n} \frac{(K_i - E)^2}{E} \tag{1}$$

gde je K_i - broj elemenata u podskupu i, a E - očekivan broj elemenata u podskupu, odnosno idealna vrednost koja je jednaka količniku ukupnog broja elemenata i broja skupova.

Iako deluje relativno komplikovano, ovo nam zapravo daje dobru ocenu koliko su (ne)ravnomerno raspoređeni elementi. U idealno slučaju bi sve K_i vrednosti bile jednake sa E i imali bi da je vrednost testa nula. U opštem slučaju se naravno javljaju odstupanja od proseka, a pošto kvadriramo udaljenosti, χ^2 brzo raste ukoliko ima većih anomalija.

Program daje vrednosti za ovaj test i u odnosu na idealan slučaj kada je broj lanaca/podskupova jednak sa brojem elemenata, kao i za realan broj podskupova (tj različitih vrednosti elemenata).

Ispitivanje različitosti dobijenih podataka

Kada se dobiju nepoznati podaci koje treba obraditi i za njih odrediti heš funkcije, nekad vredi prvo probati saznati više o različitostima dobijenih podataka.

Kod primera za korišćenje kancelarija, na primer, definisali smo da je `hashCode` zavistan samo od jednog od polja, i samim tim kad smo pokrenuli program smo saznali tačno koliko ima različitih imena i koliko ima različitih brojeva.

Isti pristup se može iskoristiti i u drugim situacijama, što nam može dati bolje ideje kako da postavimo različite koeficijente i na koja polja možda ima smisla staviti više "težine" u samom hešu.