Hash funkcije - pripremni materijal

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

2015

Contents

Heš kod i jednakost u programskom jeziku Java **1**

Ocene kvaliteta funkcije **2**

Primeri **3**

Korišćenje kancelarija 3

Gadjanje mete 6

Tabla za igru XO 8

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.

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. Prva je procentualna popunjenost tabele – odnosno procenat podskupova (listi) u kojima postoji bar po jedan element.

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.

Pri pisanju heš funkcije treba ciljati na što veću procentualnu popunjenost tabele, idealno je to 100% za veću količinu podataka.

Sa druge strane, χ^2 test predstavlja ravnomernost duzina listi, te bi trebao ostajati relativno konzistentan. Kod dobrih heš funkcija koje ravnomerno rasporedjuju elemente ovaj test često daje niže vrednosti tek za veći broj ubačenih elemenata, pošto tek onda dolaze do izražaja statističko ravnomerno raspoređivanje.

Dobre vrednosti za ovaj test za heš funkciju zavise od podataka sa kojima se radi, ali u opštem slučaju treba ciljati da budu niže od broja podskupova.

Primeri

U sledećim primerima će se razmatrati različiti tipovi podataka i kako se oni efikasno mogu skladištiti u heš tabelama. Uglavnom će se razmatrati ideje heš tabele koja ima tačno određen broj podskupova i pratiti njihovu pokrivenosti i hi kvadrat meru. Tipično će se razmatrati razlitiče veličine ovih tabela, na primer 101, 503 i 997 radi bolje ilustracije uticaja veličine na lošije heš funkcije.

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:

Elements : 4885 Percent full: 99.01 Chi square : 67.3081 ‘Columns’ : 101

Elements : 4885 Percent full: 19.88 Chi square : 19778.4749 ‘Columns’ : 503

Elements : 4885 Percent full: 10.03 Chi square : 44000.6551 ‘Columns’ : 997

Budući da ima samo 100 soba, primećujemo da vrednosti nisu zadovoljavajuće čim imamo veće tabele, odnosno imamo manje od 20% pokrivenosti tabele sa 503 mesta i oko 10% za 997 mesta.

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, čak i za relativno malu tabelu sa 101 podskupom.

```
public int hashCode() {
    return prezime.hashCode();
}
```

Elements : 4885 Percent full: 49.50 Chi square : 6320.7484 ‘Columns’ : 101

Elements : 4885 Percent full: 11.53 Chi square : 39251.8342 ‘Columns’ : 503

Elements : 4885 Percent full: 5.72 Chi square : 85190.9396 ‘Columns’ : 997

Očigledno je potrebno kombinovati ove dve vrednosti. Možemo ih jednostavno samo sabrati:

```
public int hashCode() {
    return prezime.hashCode() + broj;
}
```

Dobijamo dosta bolje rezultate u testovima:

Elements : 4885 Percent full: 100.00 Chi square : 20.7468 ‘Columns’ : 101

Elements : 4885 Percent full: 100.00 Chi square : 720.2837 ‘Columns’ : 503

Elements : 4885 Percent full: 99.00 Chi square : 1341.3007 ‘Columns’ : 997

Množenje će tipično dati bolje vrednosti nego sabiranje kad imamo vrednosti koje su jako blizu jedne drugima (što brojevi kancelarija koji idu zaredom jesu), pošto pomaže u “raspipanju” vrednosti na većem opsegu.

Na primer

```
public int hashCode() {
    return prezime.hashCode() * broj;
}
```

Elements : 4885 Percent full: 100.00 Chi square : 78.8864 ‘Columns’ : 101

Elements : 4885 Percent full: 100.00 Chi square : 483.8686 ‘Columns’ : 503

Elements : 4885 Percent full: 99.10 Chi square : 951.0727 ‘Columns’ : 997

Iz rezultata se vidi da se konzistentno dobijaju dobre vrednosti sa ovakvom heš funkcijom. Naročito je bitno što nam vrednost hi kvadrat testa govori da imamo odstupanje u veličinama podskupova u proseku manje od jedan.

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.

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.

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 je polje null pre dalje provere
        if (rezultati != null && o2.rezultati != null) {
            if (o2.rezultati.length == rezultati.length) {
                for (int i = 0; i < rezultati.length; i++) {
                    if (o2.rezultati[i] != rezultati[i]){
                        // cim je nesto razlicito nisu isti
                        return false;
                    }
                }
                // ako se sve vrednosti slazu isti su
                return true;
            }
            return false;
        } else {
            // vracamo da li su oba null, tj da li su jednaki
            return (rezultati == null && o2.rezultati == null);
```

```
        }
    }
```

"Očigledno" rešenje je samo sabrati sve ove vrednosti i vratiti broj. Međutim ovakvo rešenje daje maksimalno 200 različitih vrednosti i dosta loše vrednosti u tabeli:

Elements : 9049 Percent full: 100.00 Chi square : 314.5114 'Columns' : 101

Elements : 9049 Percent full: 27.63 Chi square : 34841.7383 'Columns' : 503

Elements : 9049 Percent full: 13.94 Chi square : 77947.1553 'Columns' : 997

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. Dodatno to možemo unaprediti množenjem prostim brojem pri svakom sabiranju, tako da se ta množenja akumuliraju na prvim elementima.

```
public int hashCode() {
    int rez = 0;
    for (int i = 0;i < rezultati.length; i++) {
        rez = (rez + rezultati[i] * (MAX_DUZ+1-i)) * 7;
    }
    return rez;
}
```

Elements : 9084 Percent full: 100.00 Chi square : 101.7743 'Columns' : 101

Elements : 9084 Percent full: 100.00 Chi square : 415.1915 'Columns' : 503

Elements : 9084 Percent full: 100.00 Chi square : 981.2660 'Columns' : 997

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 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;
}
```

Elements : 4424 Percent full: 9.90 Chi square : 87664.1198 ‘Columns’ : 101

Elements : 4424 Percent full: 1.99 Chi square : 454193.0719 ‘Columns’ : 503

Elements : 4424 Percent full: 1.00 Chi square : 904604.2717 ‘Columns’ : 997

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.

Kombinovanjem ovih ideja se mogu napraviti `hashCode` metodi koji će rezultovati sa hi kvadrat vrednostima manjim od broja podskupova (“kolona”), što se ostavlja za samostalnu vežbu.