PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET METROPOLITAN BEOGRAD

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Heširanje

Lekcija 04

PRIRUČNIK ZA STUDENTE

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Lekcija 04

HEŠIRANJE

- Heširanje
- Poglavlje 1: Uvod u heširanje
- Poglavlje 2: Formiranje heš kodova
- Poglavlje 3: Otvoreno adresiranje
- Poglavlje 4: Odvojeno ulančavanje
- Poglavlje 5: Faktor popunjenosti tabele i rehеširanje
- Poglavlje 6: Implementacija Mape u Javi
- Poglavlje 7: Implementacija seta u Javi
- Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

Uvod

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upozanju sa:

- Postupkom kreiranja heš funkcija
- Postupkom razrešavanja kolizija u toku heširanja
- Postupkom otvorenog adresiranja u cilju razrešavanja sudara
- Postupkom odvojenog ulančavanja u cilju razrešavanja kolizija
- Spoljašnjim heširanjem
- Impementacijom skupa (seta) u Javi korišćenjem heširanja
- Implementacijom mape u Javi korišćenjem heširanja.

U prethodnim lekcijama smo se bavili binarnim stablima i njihovim balansiranjem. Videli smo da je neophodno O(log n) vreme da bi pretražili element u dobro balansiranom stablu pretraživanja. Da li postoji efikasniji način za pretraživanje elementa u nekom kontejneru ? U ovoj lekciji ćemo predstaviti tehniku koja se zove heširanje. Možete da koristite heširanje da bi ste implementirali mapu ili set kod kojih se pretraživanje, ubacivanje i brisanje elemenata vrši za vreme O(1) .

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 1

Uvod u heširanje

OSNOVNI POJMOVI

Cilj heširanja je da se ubrza pretraživanje podataka u nekom skupu ili mapi, bez obilaska svih elemenata odnosno podataka u skupu

Heširanje je jedna od tehnika za pretraživanje podataka. Cilj je da se ubrza pretraživanje podataka u nekom skupu ili mapi bez obilaska svih elemenata (podataka) u skupu.

Kako funkcioniše heširanje?

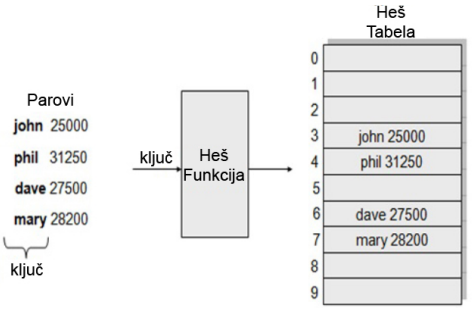
Pretpostavka je da za svaki podatak postoji ključ koji taj podatak jednoznačno određuje (npr. broj indeksa jednoznačno određuje pojedinačnog studenta). Pitanje je kako naći podatak čiji je ključ zadat.

Heširanje se može opisati na sledeći način:

- Izabere se funkcija *f(K)* koja ključ transformiše u vrednost ove funkcije
- Vrednost funkcije predstavlja ulaz u tabelu u kojoj se čuva podatak (Slika-1).

Funkcija *f (K)* naziva se heš funkcija, a tabela se naziva heš tabela. Jedan red u heš tabeli naziva se ulaz tabele.

Vrednost koju vraća heš funkcija zove se heš vrednost ili heš kod.



Slika 1.1 Primer heširanja i heš funkcije

Da rezimiramo, osnovni pojmovi kod heširanja su:

- Heš tabela: Tabela u kojoj objekat može da se nađe u O(1) vremenu. Pri tome se na osnovu ključa korišćenjem heš funkcije dobija adresa u heš tabeli.
- Heš funkcija: Funkcija koja, primenjena na ključ, daje ceo broj koji se koristi kao adresa u heš tabeli.

- Sudar (kolizija): Kada heš funkcija preslika dva različita ključa u istu adresu u tabeli, kaže se da dolazi do sudara ili kolizije.

HEŠ TABELA I HEŠ FUNKCIJA

Heš tabela (engl. hash table) ili heš mapa (engl. hash map) je struktura podataka koja koristi heš funkciju za efikasno preslikavanje određenih ključeva u njima pridružene vrednosti

Heš tabela (engl. hash table) ili heš mapa (engl. hash map) je struktura podataka koja koristi heš funkciju za efikasno preslikavanje određenih ključeva u njima pridružene vrednosti.

Na Slici-2 je prikazana heš funkcija koja preslikava imena ljudi do polja u tabeli koje čuva broj telefona. Heš funkcija se koristi za transformisanje ključa u indeks (heš) tj. mesto u nizu elemenata (heš tabeli) gde treba tražiti odgovarajuću vrednost.

Slika 1.2 a) Heš funkcija koja preslikava imena ljudi do polja u tabeli koje čuva broj telefona. b) Heš funkcija mapira ključ do odgovarajućeg indeksa u heš tabeli dimenzije N

Heš funkcije se prvenstveno koriste za generisanje izlaznih podataka fiksne dužine koji se ponašaju kao reference na originalne podatke.

Na primer, linearno pretraživanje imena osobe u listi traje duže kako se dužina liste povećava, ali heširana vrednost može skladištiti referencu na tačno jednoznačno definisano mesto, pa tako dobijamo konstantno vreme za pretragu.

Heš funkcije se koriste da bi se:

- ubrzalo traženje stavki u bazama podataka,
- izvršilo detektovanje dupliranih ili sličnih vrednosti u velikom fajlu i
- izvršilo pronalaženje sličnih segmenata u DNK sekvenci.

Kada se dva puta pozove nad identičnim podacima (npr. dva stringa koja sadrže potpuno iste karaktere) heš funkcija treba da proizvede istu vrednost. To je od ključnog značaja za ispravnost gotovo svih algoritama heširanja.

HEŠ FUNKCIJE I SUDARI KLJUČEVA

Heš funkcija je svaki algoritam koji podacima proizvoljne dužine dodeljuje podatke fiksne dužine. Vrednost koju vraća heš funkcija zove se heš vrednost ili heš kod

Osobine dobre heš funkcije:

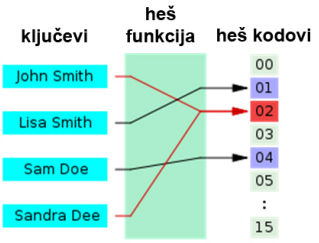
- Najbolja heš funkcija distribuira ključeve u heš tabelu što je moguće ravnomernije.
- Mora vratiti broj od *0* do *tablesize* (veličina heš tabele).
- Treba da efikasno računa – O(1) vreme.
- Ne treba nepotrebno trošiti prostor.
- Za svaki indeks, postoji najmanje jedan ključ koji se na njega heširа.

Heš funkcije obično nisu invertibilne, što znači da nije moguće rekonstruisati ulaznu vrednost *x* samo iz njene heš vrednosti *h(x)*.

Idealno, heš funkcija bi trebalo da preslikava svaki mogući ključ u zaseban indeks, ali ovaj zadatak je u praksi najčešće neostvariv.

Većina implementacija heš tabela podrazumeva da su heš kolizije (sudari – parovi različitih ključeva sa istim heš vrednostima) normalna pojava, i na neki način se stara da se ovaj problem prevaziđe (Slika-3).

U dobro dimenzionisanoj heš tabeli, prosečan broj računarskih instrukcija svakog pronalaženja je nezavisan od broja elemenata ubačenih u tabelu.



Slika 1.3 Primer upotrebe heš funkcije i kolizije (sudara) dva kluča

Uobičajeni oblik heš funkcije je:

h(v) = f(v) mod m

Naučnik Knut (Knuth- 1973) je predložio da u prethodnom izrazu vrednost *m* bude prost broj.

PRIMERI UPOTREBE HEŠIRANJA

Tipična heš funkcija prvo konvertuje ključ pretrage u neku celobrojnu vrednost koja se naziva heš kod, a zatim se vrši kompresija heš koda u indeks koji odgovara polju heš tabele

Primer 1.

Kao prost (ali nerealan) primer heširanja uzećemo smeštanje *n* podataka, pri čemu svaki od podataka ima jedinstveni ključ u opsegu od *0* do *n-1*. Podatak sa ključem *k* može biti smešten na mesto u tabeli *HT[k]*, pa je stoga heš funkcija određena kao *h(k) = k*. Da bi se pronašao podatak pomoću ključa *k*, treba pogledati polje tabele *HT[k]*.

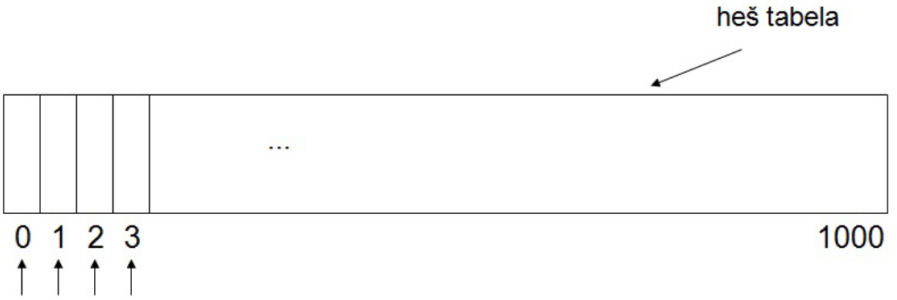
Primer 2.

Pretpostavimo da ključ može biti bilo koja vrednost u opsegu do 0 do 65,535 (tipa 16bits unsigned integer), i da je u bilo kom vremenskom trenutku u heš tabelu moguće smestiti 1000 podataka.

U tom slučaju bi bilo nepraktično koristiti heš tabelu koja ima 65,536 polja, jer bi najveći deo polja tabele ostao prazan. Stoga moramo kreirati takvu heš funkciju koja omogućuje smeštanje podataka u što manjoj heš tabeli. S obzirom da je opseg klučeva znatno veći od veličine tabele desiće se da se za bar dva ključa mapiranje vrši ka istom polju u tabeli.

Primer 3.

Potrebno je ubaciti ključeve (951000 - 952000) u tabelu sa ne više od 1000 studenata. Heš funkcija: *h(k) = k-951000* mapira brojeve u različite pozicije u tabeli 0-1000 (Slika-4).



Slika 1.4 Primer heširanja ogromnih brojeva u tabelu veličine 1000

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 2

Formiranje heš kodova

FUNKCIJA HASHCODE()

Svaka klasa u programskom jeziku Java eksplicitno ili implicitno sadrži metodu hashCode(), koja pretvara podatak instance klase u jedinstvenu heš vrednost

U programskom jeziku Java svaka klasa eksplicitno ili implicitno sadrži metodu:

hashCode()

koja pretvara podatak instance klase u jedinstvenu heš vrednost (32-bitsigned integer).

Sve klase u Javi nasleđuju osnovnu heš šemu od osnovne klase *java.lang.Object*, mada najveći deo nasleđenih klasa ima svoju predefinisanu metodu.

Osnovno pravilo za predefinisanu implementaciju heš metode je da se ponaša na način koji je konzistentan metodi *equals()*, tj da svaki pojedinačni poziv metode *equals()* nad istim objektom daje uvek isti rezultat.

Stoga, ako dva različita objekta imaju istu vrednost metode *equals()* onda moraju da imaju istu heš vrednost.

Na primer, u klasi *Zaposleni* može se implementirati heš funkcija koja se dobija slaganjem heš funkcija članova klase na sledeći način:

```
public class Zaposleni {
    int     employeeId;
    String  name;
    Department dept;

    // other methods would be in here

    @Override
    public int hashCode() {
        int hash = 1;
        hash = hash * 17 + employeeId;
        hash = hash * 31 + name.hashCode();
        hash = hash * 13 + (dept == null ? 0 : dept.hashCode());
        return hash;
    }
}
```

HEŠ KODOVI ZA PRIMITIVNE TIPOVE

Heš kodovi primitivnih tipova se kreiraju korišćenjem univerzalnog heširanja gde se slučajno bira funkcija iz familije heš funkcija sa ciljem da se ostvari što manji broj sudara ključeva

Heširanje celih brojeva

Za heširanje celih brojeva najčešće se koristi funkcija:

$h_{a,b}(x)=((ax+b) \bmod p)) \bmod m$

pri čemu je p prost broj za koji važi $p \geq u$, a a,b proizvoljno izabrani celi brojevi (*mod* je funkcija koja vraća ostatak pri deljenju dva broja).

Kao ključ pretrage za tipove ***byte***, ***short***, ***int***, i ***char***, jednostavno korititi konverziju (cast) u tip ***int*.**

Heširanje decimalnih brojeva

Za ključ koji je tipa ***float*** može se koristiti funkcija *Float.floatToIntBits(key)* kao heš kod.

Funkcija *floatToIntBits(floatf)* vraća ***int*** vrednost čija je bit reprezentacija identična bitnoj reprezentaciji float vrednosti broja *f*. Stoga, za različite ključeve brojeva tipa ***float*** dobiće se različiti heš kodovi.

Heširanje tipa long

Kao ključ za tip ***long*** može se koristiti sledeći kod:

```
int hashCode = (int)(key ^ (key >> 32));
```

Primetimo da je *>>* desni operator pomeranja bitova, koji pomera jedan bit informacija 32 pozicije u desno.

Na primer, rezultat operacije *1010110 >> 2* je *0010101*. Operator ^ je isključivo-ili operator nad bitovima. On operiše sa dva odgovarajuća bita binarnih operanada. Na primer, rezultat operacije *1010110 ^ 0110111* je *1100001*.

Heširanje tipa double

Kao ključ za tip ***double***, prvo je potrebno izvršiti konverziju u tip *long* korišćenjem funkcije *Double.doubleToLongBits*, a zatim izvršiti sklapanje i pretvaranje u ***int*** sledećim naredbama:

KOMPRESIJA HEŠ KODOVA

Heš kod nekog ključa može biti ogroman ceo broj tako da on mora biti skaliran kako bi mogao da bude u opsegu indeksa heš tabele

Heš kod nekog ključa može biti ogroman ceo broj koji je izvan opsega kada je u pitanju veličina i indeks heš tabele, tako da on mora biti skaliran kako bi mogao da bude u opsegu indeksa heš tabele.

Pretpostavimo da se indeks heš tabele kreće u opsegu od *0* do *N-1*. Najuobičajeni način skaliranja nekog velikog broja u opseg između 0 i *N*-1 je korišćenje jednačine:

h(hashCode) = hashCode % N

Da bi osigurali da je indeks ravnomerno raspoređen, poželjno je izabrati da je *N* prost broj veći od 2.

U idealnom slučaju, *N* treba da bude prost broj. Međutim, kada imamo veliki broj podataka u tabeli, malo je teže odrediti najbliži prost broj. U Java API implementaciji klase *java.util.HashMap*, izabrano je da je N broj koji je stepen broja 2. Postoji veoma dobar razlog za to. Kada je N stepen broja 2, važi da je

h(hashCode) = hashCode % N

isto kao i

h(hashCode) = hashCode & (N – 1)

Znak ampersand, *&*, je AND operator nad bitovima. Operator AND primenjen na dva bita, kao rezultat daje 1 ako su oba bita jednaka 1.

Operator *&* se izvršava znatno brže od operatora *%*.

Da bi obezbedili da je heš tabela ravnomerno raspoređena, koristi se i dodatna heš funkcija zajedno sa primarnom heš funkcijom. Dodatna funkcija je definisana kao

```
private static int supplementalHash(int h) {
 h ^= (h >>> 20) ^ (h >>> 12);
 return h ^ (h >>> 7) ^ (h >>> 4);
}
```

Operatori *^* i *>>>* su *isključivo ili* i *unsigned operator pomeranja u desno*. Operatori nad bitovima rade mnogo brže nego operatori množenja, deljenja i ostatka. Treba umesto njih koristiti operatore nad bitovima kad god je to moguće. Kompletna heš funkcija je data kao:

h(hashCode) = supplementalHash(hashCode) % N

Prethodna funkcija je ista kao i

h(hashCode) = supplementalHash(hashCode) & (N – 1)

pošto je *N* vrednost koja se dobija stepenovanjem broja *2*.

HEŠIRANJE STRINGOVA

Heš kod za string se dobija spajanjem heš kodova svakog pojedinačnog karaktera u stringu. Cilj je da heš kodovi karaktera budu jednoznačni kako bi bilo što manje sudara ključeva

Pretpostavimo da je dužina stringa označena sa *l*, i neka se string *x* sastoji iz sledećih slova **$x=(x_0,\ldots,x_l)$**. Univerzalna familija funkcija za heširanje stringova ima oblik (Slika-1):

$$h_a(\bar{x}) = h_{\text{int}}\left(\left(\sum_{i=0}^{\ell} x_i \cdot a^i\right) \bmod p\right)$$

Slika 2.1 Funkcija za heširanje stringova

gde su *a* i h_{int} proizvoljni brojevi. Pseudo-kod za prethodnu jednačinu se može napisati na sledeći način:

```
int hash(String x, int a, int p) {
 int h=x[0];
 for ( i=1 ; i < x.length ; i++)
  h = ((h*a) + x[i])) mod p;
 return h;
}
```

Poboljšana heš funkcija uključuje množenje uticaja svakog pojedinačnog karaktera stringa sa nekim faktorom, kako bi se sprečila pojava sudara (kolizija) pri heširanju permutacija istog stringa. U nastavku je primer:

```
int hash (const String key, int tableSize)
{
 int hashVal = 0;
 for (int i = 0; i < key.length; i++)
  hashVal = 37 * hashVal + key[i];
 hashVal %=tableSize;
 if (hashVal < 0)
  hashVal += tableSize;
 return hashVal;
}
```

U nastavku je data bolja implementacija heš funkcije. Ona procesira svaka 4 bajta iz stringa i interpretira taj podniz kao jedan celobrojni kod, a zatim se svi ti kodovi sabiraju i sklapaju u jedinstveni kod.

```
long sfold(String s, int M){
 int intLength = s.length() / 4;
 long sum = 0;
 for (int j = 0; j < intLength; j++) {
  char c[] = s.substring(j * 4, (j * 4) + 4).toCharArray();
```

```
  long mult = 1;
  for (int k = 0; k < c.length; k++){
   sum += c[k] * mult;
   mult *= 256;
  }
 }
 char c[] = s.substring(intLength * 4).toCharArray();
 long mult = 1;
 for (int k = 0; k < c.length; k++) {
  sum += c[k] * mult;
  mult *= 256;
 }
 return(Math.abs(sum) % M);
}
```

PRIMER PROSTOG HEŠIRANJA

Skaliranje heš koda se vrši korišćenjem funkcije moduo (ostatak pri deljenju), pri čemu se određuje ostatak pri deljenju ključa sa vrednošću koja predstavlja veličinu heš tabele

Pretpostavimo da su studenti nekog fakulteta označeni nekim ID-om koji je u opsegu 1-1000000. Neka se heš tabela sastoji iz 1600 polja. U cilju transformisanja ključa u odgovarajući indeks tabele koristimo heš funkciju oblika: *hash (key)*.

Neka je funkcija definisana na sledeći (najprostiji mogući) način:

```
hashKey = studentId % 1600;
// a simple in-line hash "function"
```

Koristimo operaciju *ostatak pri deljenju* da bi smo podatak koji se sastoji iz maksimalno 6 cifara smestili u tabelu dimenzije 1600 (konvertovanje ključa da bi se smestio u opseg 0-1599).

Jedan od mogućih načina ubacivanja novog podatka u heš tabelu je korišćenje funkcije *add* čiji je kod dat u nastavku:

```
public void add (Hashable element)
// Adds element to this list at position element.hash().
{
  int location;
  location = element.hash();
  list[location] = element;
  numElements++;
}
```

Da bi pristupili podatku koji je prethodno pomenutom funkcijom heširan u tabelu, možemo koristiti funkciju *get*:

```
public Hashable get(Hashable element)
   // Returns an element e from this list such
   // that e.equals(element).
{
   int location;
   location = element.hash();
   return (Hashable)list[location];
}
```

Ovo je samo jedan ilustrativni primer koji pokazuje princip heširanja.

Naravno, za pravilan rad heširanja i pojavu što manje kolizija potrebno je izvršiti znatna poboljšanja i uvesti nove pretpostavke kod heš funkcija, i to će biti opisano u nastavku.

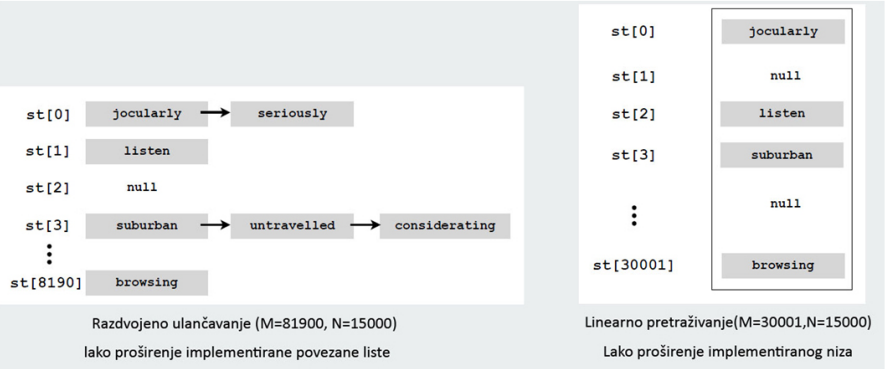
RAZREŠAVANJE KOLIZIJA (SUDARA)

Postoje dva pristupa u razrešavanju kolizija: odvojeno ulančavanje (korišćenjem povezanih listi za svako polje u tabeli) i otvoreno adresiranje (korišćenjem nizova za tabelu)

Postoje dva pristupa u razrešavanju kolizija:

1. *Odvojeno ulančavanje* - Ubaciti ključeve koji se sudaraju u jedinstvenu povezanu listu koja odgovara tom indeksu (Slika-2 levo)

2. *Otvoreno adresiranje* - Kada se novi ključ sudara sa nekim koji je već ubačen u tabelu, pronaći sledeće slobodno mesto u tabeli i smestiti ga tu (Slika-2 desno).



Slika 2.2 Razrešavanje kolizije a) odvojenim ulančavanjem, b) otvorenim adresiranjem uz pomoć linearne probe

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Šta je heš kod? Šta je haš kod za sledeće tipove: Byte, Short, Integer, and Character?

Zadatak 2. Kako se računa heš kod za Float objekat?

Zadatak 3. Kako se računa heš kod za Long objekat?

Zadatak 4. Kako se računa heš kod za Double objekat?

Zadatak 5. Kako se računa heš kod za String objekat?

Zadatak 6. Kako se heš kod kompresuje u ceo broj koji prestavlja indeks u heš tabeli?

Zadatak 7. Ako je N stepen dvojke, da li je N/2 isto što i N >> 1? Zašto?

Zadatak 8. Ako je N stepen dvojke, da li je m % N isto što i m & (N – 1) za bilo koji ceo broj m? Zašto?

Poglavlje 3

Otvoreno adresiranje

UVOD U OTVORENO ADRESIRANJE

Ideja razrešavanja kolizije kod otvorenog adresiranja je da se podatak koji je u koliziji smesti na neko drugo mesto u tabeli koje nije zauzeto

Jedan od načina da se razreši kolizija je da se podatak koji je u koliziji smesti na neko drugo mesto u tabeli koje nije zauzeto. Ovako smeštanje podataka u heš tabelu se naziva otvoreno adresiranje.

Karakteristike otvorenog adresiranja su:

- ne koristi se povezana lista za svako polje tabele
- svi element su uskladišteni u tabeli
- jedan element po polju tabele
- heš funkcija definiše redosled probanja u cilju smestanja ključa.

Kod otvorenog adresiranja, svaka pozicija u tabeli je u jednom od tri stanja *PRAZNO*, *OBRISANO* ili *ZAUZETO*:

- U slučaju da je polje *ZAUZETO* ono sadrži legitimne vrednosti (ključ i podatak). U suprotnom ne sadrži nikakvu vrednost, tj. polje je *PRAZNO*.
- Ukoliko se sa nekog polja obriše vrednost takvo polje će biti označeno sa *OBRISANO*, a ne *PRAZNO*.

Algoritam za ubacivanje nove vrednosti u heš tabelu se sastoji iz sledećih koraka:

- **1.** Izračunati vrednost indeksa u heš tabeli na kome treba smestiti novu vrednost V, korišćenjem heš funkcije $P = H(key(V))$.
- **2.** Ukoliko polje nije *ZAUZETO* smestiti vrednost V u to polje.
- **3.** Ukoliko je polje P *ZAUZETO*, izračunati novo polje u tabeli korišćenjem heš funkcije P a zatim ponoviti korake (2) i (3).

Bilo koji metod razrešavanja kolizije možemo posmatrati kao generisanje niza polja heš tabele koja su slobodna i na kojima se potencijalno može smestiti podatak. Prvo polje u sekvenci može biti originalno polje dobijeno pomoću heš funkcije. Ukoliko je ono zauzeto odlazi se do sledećeg polja u sekvenci, pa ako je i to polje zauzeto onda se ide dalje i tako redom.

Ova sekvenca potencijalnih polja se naziva *sekvenca pretrage*, i ona se generiše korišćenjem neke od *probnih funkcija*.

Postoji nekoliko različitih načina konstruisanja probne funkcije. U nastavku će biti opisane sledeće metode:

- *Linearna pretraga*
- *Kvadratna pretraga*
- *Dvostruko heširanje*.

PRIMER RAZREŠAVANJA SUDARA

Funkcija heširanja prvo proverava da li je osnovno polje ključa prazno. Ukoliko je polje zauzeto koristimo funkciju pretrage sa ciljem da se locira prvo slobodno polje u tabeli

Pretpostavimo da je **probna funkcija** označena sa *p*, i u tom slučaju ubacivanje novog podatka u heš tabelu može biti opisano sledećim kodom:

```
// Insert e into hash table HT
public void hashInsert(const Key k, const Elem e)
{
 int home;                  // Home position for e
 int pos = home = h(k);     // Init probe sequence
 for (int i=1; EMPTYKEY != (HT[pos]).key(); i++)
 {
  pos = (home + p(k, i)) % M; // probe
  if (k == HT[pos].key())
  {
   // print, "Duplicates not allowed\n";
   return;
  }
 }
 HT[pos] = e;
}
```

Funkcija *hashInsert* prvo proverava da li je osnovno polje ključa prazno. Ukoliko je polje zauzeto koristimo funkciju pretrage *p(k, i)* sa ciljem da se locira prvo slobodno polje u tabeli.

Funkcija *p* ima dva parametra: ključ *k* i brojač *i* koji kojim se prolazi kroz članove sekvence probne funkcije *p*. Stoga, ako želimo da pristupimo prvom članu probne sekvence ključa pozivamo *p(K,1)*, drugom članu pristupamo sa *p(K,2)* i tako redom.

Napomena: Probna funkcija vraća rastojanje (offset) od originalne (osnovne) pozicije ključa a veoma retko indeks polja heš tabele. Stoga, ***for*** petlja u funkciji *hashInsert* sračunava sledeću probnu poziciju u tabeli tako što dodaje vrednost probne funkcije na indeks osnovnog polja ključa *K*.

Za pretraživanje heš tabele se koristi ista probna sekvenca koja je korišćena za ubacivanje novih elemenata u tabelu.

Funkcije ubacivanja i pretrage, istovremeno, rade pod pretpostavkom da je bar jedno od polja u probnoj sekvenci ključa *K* prazno i da na njega možemo smestiti podatak. U suprotnom, javiće se beskonačna petlja a samim tim i neuspešno pretraživanje.

Stoga, sistem heširanja mora imati podatak o trenutnom broju elemenata u tabeli i voditi računa o tome da heš tabela ne ostane bez praznih mesta.

3.1 Linearna pretraga

OSNOVI LINEARNE PRETRAGE

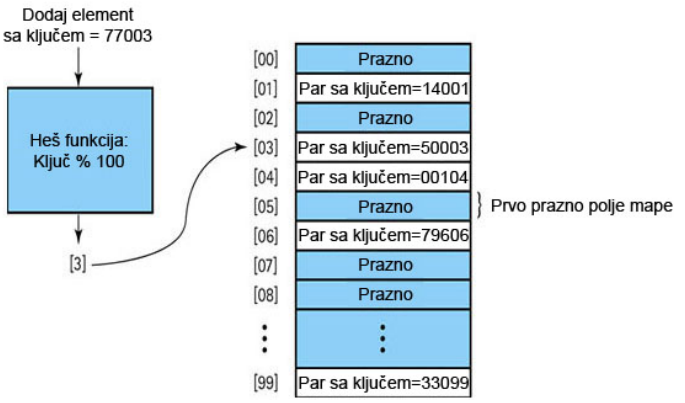
Ukoliko se javi kolizija ključeva, tj. ako se desi da je polje već popunjeno, linearna proba ispituje svako sledeće polje u heš tabeli dok ne naiđe na prvo slobodno i tu smešta ulaz mape

Linearna pretraga radi po sledećem principu:

- Za odgovarajući podatak sa ključem *key*, izračunava se indeks u tabeli korišćenjem heš funkcije *hash(key)*.
- Ukoliko polje u tabeli *data[hash(key)]* ne sadrži nikakav podatak (prazno polje), onda smestiti podatak u polje *data[hash(key)]* i završiti algoritam ubacivanja.
- Ukoliko je lokacija *data[hash(key)]* zauzeta onda probati sa sledećim poljem, tj. *data[hash(key)+1]*. Ukoliko je i to polje zauzeto proveriti *data[hash(key)+2]*, i tako redom, dok se ne pronađe prvo slobodno polje. Kada se stigne do poslednjeg člana heš tabele vraćamo se na početak (pozicija 0), i nastavljamo sa pretragom dok se ne dođe do prvog slobodnog polja.

Primer 1.

Neka je u tabelu sa Slike-1, koja ima 100 polja, već ubačeno nekoliko ključeva, i neka je sledeći ključ koji treba ubaciti označen sa *77003*. Koristimo standardnu heš funkciju *h(k) = k % 100* za određivanje koda. Dobijen je kod 3 ali, pošto je to polje zauzeto isprobavamo redom sledeća polja (4 i 5) i konačno ubacujemo ključ na prvo slobodno polje u tabeli.



Slika 3.1.1 Linearna proba: ubacivanje ključa 77003 u heš tabelu

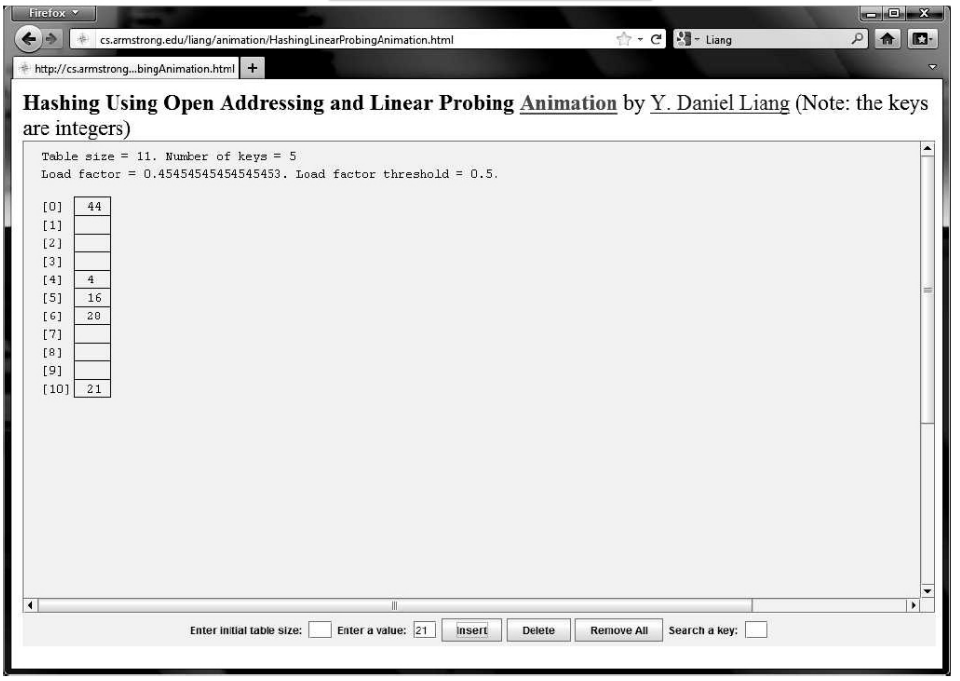
Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ANIMACIJA LINEARNE PRETRAGE

Aplikacija sa animacijom linearne pretrage radi boljeg uvida u postupak

Na sledećem linku se nalazi interaktivna aplikacija koja pokazuje kako radi linearna pretraga kod rešavanja kolizija: www.cs.armstrong.edu/liang/animation/HashingLinearProbingAnimation.html

Izgled aplikacije je prikazan na **Slika-2.**



Slika 3.1.2 Animacija koja prikazuje kako radi linearna pretraga kod kolizije ključeva

PRIMER LINEARNE PRETRAGE I POJAVA KLASTERA

Najveća mana linearne pretrage je pojava klastera elemenata usled ogromnog broja ključeva sa istim heš kodom što povećava vreme ubacivanja podataka a i pretraživanja podataka u tabeli

Funkcija linearne probe je data kao: *P = (1 + P) mod TABLE_SIZE*. Neka je veličina heš tabele *TABLE_SIZE=10*, ključevi su dvocifreni celi brojevi a heš funkcija oblika *H(k) = k mod 10*. Neka su inicijalno sve pozicije u heš tabeli prazne (Slika-3):

0	1	2	3	4	5	6	7	8	9
empty	empty	empty	empty	empty	empty	empty	empty	empty	empty

Slika 3.1.3 Prazna heš tabela

Ukoliko za početak u tabelu ubacimo brojeve 15, 17, i 8, neće doći do kolizije i rezultat je (Slika-4):

0	1	2	3	4	5	6	7	8	9
empty	empty	empty	empty	empty	15	empty	17	8	empty

Slika 3.1.4 Ubacivanje brojeva 15, 17 i 8 bez sudara

Kada u tabelu ubacujemo broj 35, prvo izračunamo heš kod preko *P = H(35) = 35 mod 10 = 5*. Pozicija u tabeli 5 je već zauzeta (tu je 15), pa moramo tražiti slobodno polje u tabeli. Prva sledeća pozicija na kojoj tražimo je *5+1 mod 10 = 6*. Pozicija 6 je prazna pa se broj 35 ubacuje na to polje (Slika-5).

Sada ubacujemo broj 25. Prvo probamo sa pozicijom *25 mod 10 = 5*, koja je naravno zauzeta.

0	1	2	3	4	5	6	7	8	9
empty	empty	empty	empty	empty	15	35	17	8	empty

Slika 3.1.5 Heš kod broja 35 se sudara sa ključem broja 15. Broj 35 se pomera na prvo sledeće polje

Sledeće polje na kome tražimo je 6 (*5+1 mod 10*), koje je takođe zauzeto. Zatim proveravamo pozicije 7 (*6+1 mod 10*), pa 8 (*7+1 mod 10*) koje su takođe zauzete. Sledeća je pozicija 9 (*8+1 mod 10*), koja je prazna, pa broj 25 ubacujemo na to mesto (Slika-6).

0	1	2	3	4	5	6	7	8	9
empty	empty	empty	empty	empty	15	35	17	8	25

Slika 3.1.6 Ubacivanje broja 25. Sudar sa ključem broja 15, u razrešavanje sudara ubacivanjem broja 5 u prvo slobodno polje

Sledeći broj koji ubacujemo je 75. Prvo polje koje proveravamo je 5, a zatim redom polja 6, 7, 8 i 9, koja su takođe zauzeta. Konačno, proveravamo poziciju 0 (9+1 mod 10) koja je prazna pa 75 smeštamo na to mesto (Slika-7).

0	1	2	3	4	5	6	7	8	9
75	empty	empty	empty	empty	15	35	17	8	25

Slika 3.1.7 Ubacivanje broja 75. Sudar sa ključem broja 15, i pojava klastera

Šta ako u tabelu ubacimo brojeve 85, pa 95 i 55 ?

Javiće se klasteri, i to je jedan do problema linearne pretrage.

PRIMER: LINEARNA PRETRAGA

Cilj zadatka je implementacija heš tabele koja koristi princip linearne pretrage za razrešavanje kolizije ključeva

U nastavku je dat primer implementacije heš tabele:

```
import java.util.Scanner;

class LinearProbingHashTable {
 private int currentSize, maxSize;
```

```
private String[] keys;
private String[] vals;

public LinearProbingHashTable(int capacity){
 currentSize = 0;
 maxSize = capacity;
 keys = new String[maxSize];
 vals = new String[maxSize];
}

public void makeEmpty(){
 currentSize = 0;
 keys = new String[maxSize];
 vals = new String[maxSize];
}

public int getSize() {
 return currentSize;
}

public boolean isFull() {
 return currentSize == maxSize;
}

public boolean isEmpty() {
 return getSize() == 0;
}

public boolean contains(String key) {
 return get(key) !=  null;
}

private int hash(String key){
 return key.hashCode() % maxSize;
}

public void insert(String key, String val){
 int tmp = hash(key);
 int i = tmp;
 do{
  if (keys[i] == null){
   keys[i] = key;
   vals[i] = val;
   currentSize++;
   return;
  }
  if (keys[i].equals(key)) {
   vals[i] = val;
   return;
  }
  i = (i + 1) % maxSize;
 } while (i != tmp);
}
```

```
 public String get(String key){
  int i = hash(key);
  while (keys[i] != null){
   if (keys[i].equals(key))
    return vals[i];
   i = (i + 1) % maxSize;
  }
  return null;
 }

 public void remove(String key){
  if (!contains(key))
   return;

  /** find position key and delete **/
  int i = hash(key);
  while (!key.equals(keys[i]))
   i = (i + 1) % maxSize;
  keys[i] = vals[i] = null;

  /** rehash all keys **/
  for (i = (i + 1) % maxSize; keys[i] != null; i = (i + 1) % maxSize){
   String tmp1 = keys[i], tmp2 = vals[i];
   keys[i] = vals[i] = null;
   currentSize--;
   insert(tmp1, tmp2);
  }
  currentSize--;
 }

 public void printHashTable(){
  System.out.println("\nHash Table: ");
  for (int i = 0; i < maxSize; i++)
   if (keys[i] != null)
    System.out.println(keys[i] +" "+ vals[i]);
  System.out.println();
 }
}
```

Kreirana klasa može biti istestirana korišćenjem sledećeg koda:

```
public class LinearProbingHashTableTest{
 public static void main(String[] args){
  Scanner scan = new Scanner(System.in);
  System.out.println("Hash Table Test\n\n");
  System.out.println("Enter size");
  /** maxSizeake object of LinearProbingHashTable **/
  LinearProbingHashTable lpht = new LinearProbingHashTable(scan.nextInt() );

  char ch;
  /**  Perform LinearProbingHashTable operations  **/
  do{
```

```
   System.out.println("\nHash Table Operations\n");
   System.out.println("1. insert ");
   System.out.println("2. remove");
   System.out.println("3. get");
   System.out.println("4. clear");
   System.out.println("5. size");

   int choice = scan.nextInt();
   switch (choice){
   case 1 :
    System.out.println("Enter key and value");
    lpht.insert(scan.next(), scan.next() );
    break;
   case 2 :
    System.out.println("Enter key");
    lpht.remove( scan.next() );
    break;
   case 3 :
    System.out.println("Enter key");
    System.out.println("Value = "+ lpht.get( scan.next() ));
    break;
   case 4 :
    lpht.makeEmpty();
    System.out.println("Hash Table Cleared\n");
    break;
   case 5 :
    System.out.println("Size = "+ lpht.getSize() );
    break;
   default :
    System.out.println("Wrong Entry \n ");
    break;
   }
   lpht.printHashTable();

   System.out.println("\nDo you want to continue (Type y or n) \n");
   ch = scan.next().charAt(0);
  } while (ch == 'Y'|| ch == 'y');
 }
}
```

3.2 Kvadratna pretraga

OSNOVI KVADRATNE PRETRAGE

*Jedan od načina da se eliminiše grupisanje elemenata (klaster) je korišćenje kvadratnog pretraživanja. Probna funkcija kvadratnog pretraživanja ima oblik: p(K,i) = c1*i^2 + c2*i + c3*

Jedan od načina da se eliminiše grupisanje elemenata (klaster) je korišćenje kvadratnog pretraživanja. Probna funkcija kvadratnog pretraživanja ima oblik:

$$\mathbf{p(K,i) = c_1 * i^2 + c_2 * i + c_3}.$$

Najprostiji oblik funkcije je $\mathbf{p(K,i) = c_1 * i^2}$ pa ćemo u nastavku analizirati primer kod koga se probna sekvenca definiše na sledeći način:

$$\mathbf{address = (h(key) + c\ i^2)\ \%\ tablesize}$$

Ukoliko je c =1, probna sekvenca će se sastojati iz sledećih članova:

0 *(h(key) + 0) % tablesize*

1 *(h(key) + 1) % tablesize*

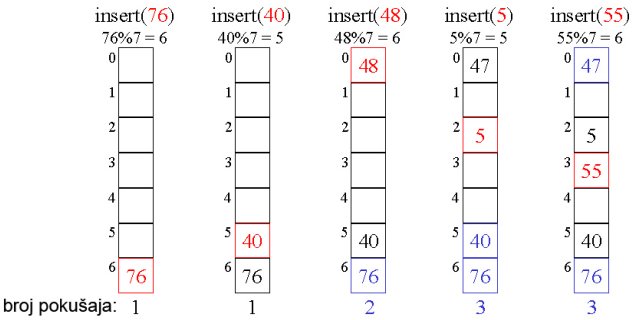
2 *(h(key) + 4) % tablesize*

3 *(h(key) + 9) % tablesize*

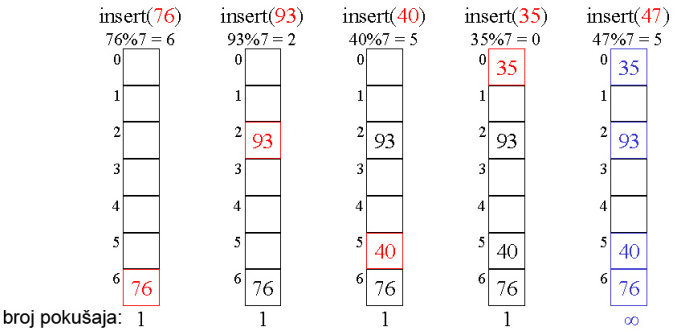
...

Posmatrajmo dva primera gde se ubacuju ključevi u heš tabelu korišćenjem prethodno definisane funkcije pretrage. Prvi primer je umetanje brojeva 76,40,48, 5 i 55 (Slika-1).

Drugi primer je umetanje brojeva 76, 93, 40,35 i 47 (Slika-2).



Slika 3.2.1 Ubacivanje ključeva 76,40,48, 5 i 55



Slika 3.2.2 Ubacivanje ključeva 76, 93, 40,35 i 47

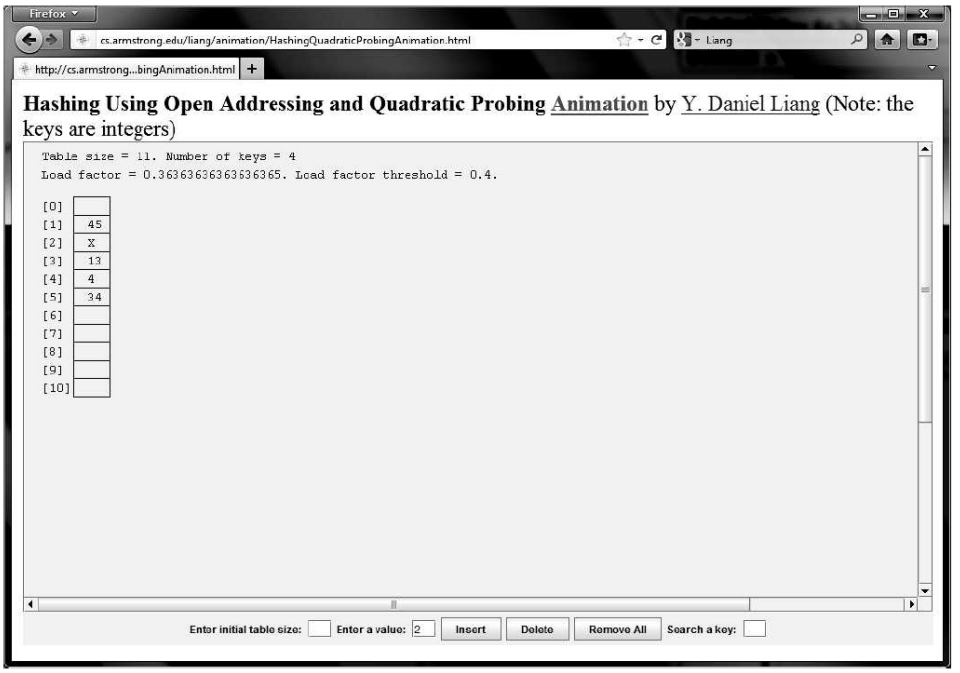
Kao što se može videti sa Slike 2, pri ubacivanju koda 47 u heš tabelu ući ćemo u beskonačnu petlju tako da je neophodno koristiti neki drugi vid razrešavanja kolizije. Problem koji je nastao se naziva sekundarni klasering.

ANIMACIJA KVADRATNE PROBE

Sticanje boljeg uvida u postupak kvadratne probe uz animaciju heširanja

Na sledećem linku se nalazi interaktivna aplikacija koja pokazuje kako radi kvadratna pretraga kod rešavanja kolizija:

www.cs.armstrong.edu/liang/animation/HashingQuadraticProbingAnimation.html,
Izgled aplikacije je prikazan na **Slika-3.**



Slika 3.2.3 Animacija koja prikazuje kako radi kvadratna pretraga kod kolizije ključeva

PRIMER: KVADRATNA PRETRAGA

Cilj zadatka je implementacija heš tabele koja koristi princip kvadratne pretrage za razrešavanje kolizije ključeva

Implementacija klase za heš tabelu:

```
import java.util.Scanner;
class QuadraticProbingHashTable{
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity)   {
        currentSize = 0;
        maxSize = capacity;
```

```
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public void makeEmpty()   {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public int getSize()    {
        return currentSize;
    }

    public boolean isFull()   {
        return currentSize == maxSize;
    }

    public boolean isEmpty()   {
        return getSize() == 0;
    }

    public boolean contains(String key)   {
        return get(key) !=  null;
    }

    private int hash(String key)   {
        return key.hashCode() % maxSize;
    }

    public void insert(String key, String val)   {
        int tmp = hash(key);
        int i = tmp, h = 1;
        do {
            if (keys[i] == null) {
                keys[i] = key;
                vals[i] = val;
                currentSize++;
                return;
            }
            if (keys[i].equals(key))  {
                vals[i] = val;
                return;
            }
            i = (i + h * h++) % maxSize;
        } while (i != tmp);
    }

    public String get(String key)  {
        int i = hash(key), h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key))
                return vals[i];
```

```
            i = (i + h * h++) % maxSize;
            System.out.println("i "+ i);
        }
        return null;
    }

    public void remove(String key) {
        if (!contains(key))
            return;

        /** find position key and delete **/
        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + h * h++) % maxSize;
        keys[i] = vals[i] = null;

        /** rehash all keys **/
        for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) %
maxSize)  {
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }

    public void printHashTable()  {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++)
            if (keys[i] != null)
                System.out.println(keys[i] +" "+ vals[i]);
        System.out.println();
    }
}
```

Test program za ispitivanje funkcionalnosti kreirane klase:

```
public class QuadraticProbingHashTableTest {
    public static void main(String[] args)  {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");
        /** maxSizeake object of QuadraticProbingHashTable **/
        QuadraticProbingHashTable qpht = new
QuadraticProbingHashTable(scan.nextInt() );

        char ch;
        /**  Perform QuadraticProbingHashTable operations  **/
        do   {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
```

```
            System.out.println("3. get");
            System.out.println("4. clear");
            System.out.println("5. size");

            int choice = scan.nextInt();
            switch (choice) {
            case 1 :
                System.out.println("Enter key and value");
                qpht.insert(scan.next(), scan.next() );
                break;
            case 2 :
                System.out.println("Enter key");
                qpht.remove( scan.next() );
                break;
            case 3 :
                System.out.println("Enter key");
                System.out.println("Value = "+ qpht.get( scan.next() ));
                break;
            case 4 :
                qpht.makeEmpty();
                System.out.println("Hash Table Cleared\n");
                break;
            case 5 :
                System.out.println("Size = "+ qpht.getSize() );
                break;
            default :
                System.out.println("Wrong Entry \n ");
                break;
            }
            qpht.printHashTable();

            System.out.println("\nDo you want to continue (Type y or n) \n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y'|| ch == 'y');
    }
}
```

3.3 Dvostruko heširanje

OSNOVI DVOSTRUKOG HEŠIRANJA

Ideja kod dvostrukog heširanja je da se koriste dve funkcije: primarna funkcija za određivanje polja u tabeli, i sekundarna funkcija kojom se određuje inkerement probnog indeksa

Jedan od veoma prostih načina da se umanji grupisanje (klaster) je korišćenje nekog izraza da se uveća korak probe P, ali da taj izraz nije konstanta kao kod linearne funkcije već da je to neka vrednost koja zavisi od ključa. Stoga ćemo umesto linearne funkcije priraštaja:

P = (1 + P) mod TABLE_SIZE

koristiti sledeću funkciju kao probnu:

P = (P + INCREMENT(Key)) mod TABLE_SIZE

Ova tehnika se naziva dvostruko heširanje.

Pretpostavimo da je u našem primeru priraštaj definisan kao

INCREMENT(Key) = 1 + (Key mod 7)

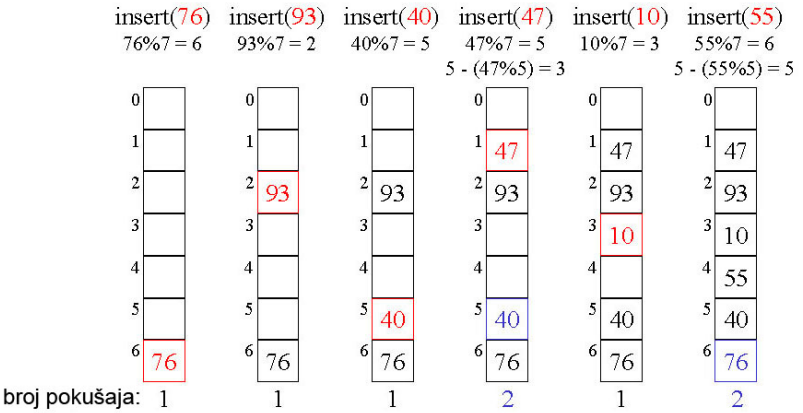
pri čemu dodajemo 1 kako bi obezbedili tako da priraštaj nikad nije jednak nuli.

Primer

Neka je sekundarna funkcija definisana kao:

Hash2(x) = R – (x % R)

U tom slučaju će postupak ubacivanja brojeva 76, 93, 40,47,10 i 55 u heš tabelu izgledati kao na Slici-1.



Slika 3.3.1 Primer dvostrukog heširanja

POSTUPAK DVOSTRUKOG HEŠIRANJA

Sekundarna funkcija kod dvostrukog heširanja određuje inkrement sledećeg indeksa u probnoj sekvenci. Pri tome treba obezbediti da priraštaj nikad nije jednak nuli

Neka su definisane primarna (***h***) i sekundarna (***h'***) heš funkcija za ubacivanje elemenata u heš tabelu dimenzije 11 na sledeći način:

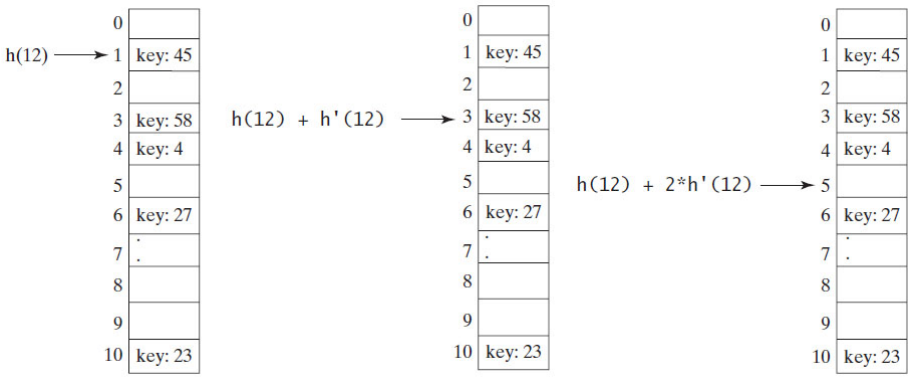
```
h(k) = k % 11;
h'(k) = 7 – k % 7;
```

Pretpostavimo da u tabeli već imamo ubačene vrednosti i da je sledeći ključ koji treba ubaciti broj 12. Kao rezultat prethodnih izraza, imaćemo sledeće vrednosti primarne i sekundarne funkcije.

```
h(12) = 12 % 11 = 1;
h'(k) = 7 – 12 % 7 = 2;
```

Probna sekvenca za ključ 12 počinje od indeksa 1 i ima priraštaj 2, kao što je pokazano na Slici-2.

Na ovaj način biće kreirana sledeća probna sekvenca: 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10 koja, kao što vidimo, obuhvata sva polja heš tabele.



Slika 3.3.2 Sekundarna funkcija kod dvostrukog heširanja određuje inkrement sledećeg indeksa u probnoj sekvenci

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER: DVOSTRUKO HEŠIRANJE

Cilj zadatka je implementacija heš tabele koja koristi primarnu funkciju za određivanje polja u heš tabeli, i sekundarnu funkcije za određivanje inkrementa pri traženju slobodnog polja

U nastavku je kod u programskom jeziku Java za implementaciju heš mape korišćenjem dvostrukog heširanja:

```
import java.io.IOException;

public class HashTableWithDoubleHashing {
  private DataItem[] hashArray;

  private int arraySize;

  private DataItem bufItem; // for deleted items

  HashTableWithDoubleHashing(int size) {
```

```
    arraySize = size;
    hashArray = new DataItem[arraySize];
    bufItem = new DataItem(-1);
  }

  public void displayTable() {
    System.out.print("Table: ");
    for (int j = 0; j < arraySize; j++) {
    if (hashArray[j] != null)
        System.out.print(hashArray[j].getKey() + " ");
    else
        System.out.print("** ");
    }
    System.out.println("");
  }

  public int hashFunc1(int key) {
    return key % arraySize;
  }

  public int hashFunc2(int key) {
    return 6 - key % 6;
  }

  public void insert(int key, DataItem item) {
    int hashVal = hashFunc1(key); // hash the key
    int stepSize = hashFunc2(key); // get step size
    // until empty cell or -1
    while (hashArray[hashVal] != null &amp;&amp; hashArray[hashVal].getKey() != -1)
{
    hashVal += stepSize; // add the step
    hashVal %= arraySize; // for wraparound
    }
    hashArray[hashVal] = item; // insert item
  }

  public DataItem delete(int key) {
    int hashVal = hashFunc1(key);
    int stepSize = hashFunc2(key); // get step size

    while (hashArray[hashVal] != null) {
    if (hashArray[hashVal].getKey() == key) {
        DataItem temp = hashArray[hashVal]; // save item
        hashArray[hashVal] = bufItem; // delete item
        return temp; // return item
    }
    hashVal += stepSize; // add the step
    hashVal %= arraySize; // for wraparound
    }
    return null; // can't find item
  }

  public DataItem find(int key) {
```

```
    int hashVal = hashFunc1(key); // hash the key
    int stepSize = hashFunc2(key); // get step size

    while (hashArray[hashVal] != null) {
    if (hashArray[hashVal].getKey() == key)
        return hashArray[hashVal]; // yes, return item
    hashVal += stepSize; // add the step
    hashVal %= arraySize; // for wraparound
    }
    return null; // can't find item
  }

  public static void main(String[] args) throws IOException {
    int aKey;
    DataItem aDataItem;
    int size, initSize;
    size = 100;
    initSize = 10;
    HashTableWithDoubleHashing theHashTable = new HashTableWithDoubleHashing(
        size);
    for (int i = 0; i < initSize; i++) {
    aKey = (int) (java.lang.Math.random() * 2 * size);
    aDataItem = new DataItem(aKey);
    theHashTable.insert(aKey, aDataItem);
    }

    theHashTable.displayTable();
    aKey = 100;
    aDataItem = new DataItem(aKey);
    theHashTable.insert(aKey, aDataItem);
    aKey = 100;
    theHashTable.delete(aKey);
    aKey = 100;
    aDataItem = theHashTable.find(aKey);
    if (aDataItem != null)
    System.out.println("Found " + aKey);
    else
    System.out.println("Could not find " + aKey);
  }

}

class DataItem {
  private int data;

  public DataItem(int i) {
    data = i;
  }

  public int getKey() {
    return data;
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke:

Zadatak 1. Šta je otvoreno adresiranje? Šta je linearna pretraga? Šta je kvadratna pretraga? Šta je dvostruko heširanje?

Zadatak 2. Opisati problem klasterovanja kod linearne pretrage.

Zadatak 3. Šta je sekundarno klasterovanje?

Zadatak 4. Prikazati heš tabelu veličine 11 posle unosa elemenata sa ključevima 34, 29, 53, 44, 120, 39,45 i 40 upotrebom linearne pretrage.

Zadatak 5. Prikazati heš tabelu veličine 11 posle unosa elemenata sa ključevima 34, 29, 53, 44, 120, 39,45 i 40 upotrebom kvadratne pretrage.

Zadatak 6. Prikazati heš tabelu veličine 11 posle unosa elemenata sa ključevima 34, 29, 53, 44, 120, 39,45 i 40 upotrebom dvostrukog heširanja sa sledećim funkcijama:

```
h(k) = k % 11;
h'(k) = 7 – k % 7;
```

Poglavlje 4

Odvojeno ulančavanje

OSNOVI ODVOJENOG ULANČAVANJA

Prema metodi odvojenog ulančavanja, svaka pozicija u heš tabeli sadrži listu neograničene dužine u koju će biti smešteni podaci sa istim heš kodom

Otvoreno adresiranje ima tu prednost da je memorija sa kojom radi fiksne dužine. Međutim, ono ima sledeća dva nedostatka:

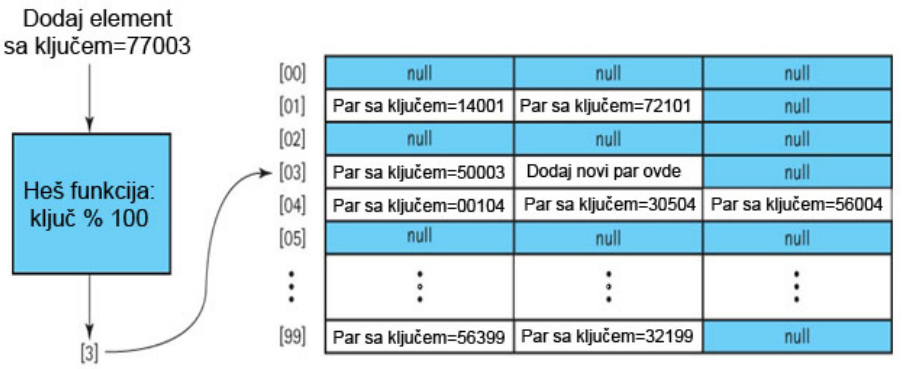
- teška reprodukcija ključa kada se ključ ne nalazi u tabeli.
- broj podataka koji se može smestiti u tabelu je ograničen, i što je još gore, kada nivo ispunjenosti tabele postaje veći dolazi do sve težeg pronalaženja slobodnih polja korišćenjem neke od probnih funkcija.

Jedan on načina da se nadomeste ovi nedostaci je korišćenje tehnike ulančavanja.

Prema ovoj metodi, svaka pozicija u heš tabeli sadrži listu neograničene dužine u koju će biti smešteni podaci sa istim heš kodom.

Stoga, ukoliko želimo da ubacimo ključ *K* u heš tabelu, mi ga dodajemo kolekciji koja se nalazi na polju *H(K)*. Slično tome, ukoliko želimo da obrišemo vrednost čiji je ključ *K*, mi je jednostavno brišemo iz kolekcije polja *K*.

Kolekcija na odgovarajućem polju heš tabele se najčešće formira korišćenjem povezane liste koje ustvari ima oblik lanca, pa se stoga i metoda naziva odvojeno ulančavanje. Primer je dat na Slici-1



Slika 4.1 Primer odvojenog ulančavanja (ulančano heširanje)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

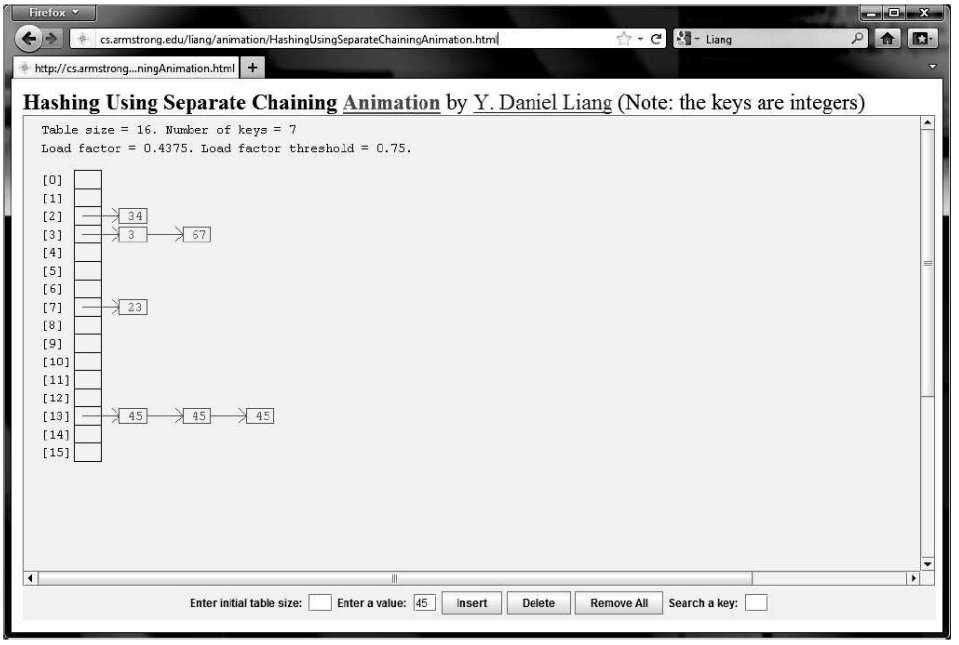
ANIMACIJA ODVOJENOG ULANČAVANJA

Sticanje boljeg uvida u odvojeno ulančavanje uz animaciju heširanja

Na sledećem linku se nalazi interaktivna aplikacija koja pokazuje kako radi odvojeno ulančavanje:

www.cs.armstrong.edu/liang/animation/HashingUsingSeparateChainingAnimation.html

Izgled aplikacije je prikazan na **Slika-2**.



Slika 4.2 Animacija koja prikazuje kako radi odvojeno ulančavanje.

PRIMER ODVOJENOG ULANČAVANJA

Kolekcija na odgovarajućem polju heš tabele se najčešće formira korišćenjem povezane liste koje ustvari ima oblik lanca, pa se stoga i metoda naziva ulančavanje

U nastavku je dat primer odvojenog ulančavanja. Kreirana je klasa *LLNode* koja će da služi kao polje u heš tabeli i definisana je kao čvor jednostruko povezane liste. Heš tabela *HashTable[]* je definisana kao niz čvorova tipa *LLNode*.

```
private class LLNode {
   private Object value;
   private LLNode   next;
   private LLNode(Object value, LLNode next){
```

```
    this.value = value;
    this.next = next;
   }
}
final static int SIZE = 127;
LLNode HashTable[] = new LLNode[SIZE];
```

Neka je ključ dat u obliku stringa tako da koristimo sledeću heš funkciju za pretvaranje ključa u odgovarajući indeks heš tabele:

```
private int Hash(){
// as an example, this hash function folds strings
    int sum = 0;
    for(int i=0; i<key.length; i++){
    sum = sum + key.charAt(i)*(key.length-i);//weighted
    }
    return sum % SIZE;
}
```

Funkcije za ubacivanje novog ključa i pristupanje elementu heš tabele preko ključa mogu biti implementirane na sledeći način:

```
public static void add (Hashable element)
// Adds element to this list at position element.hash(),
// or the next free array slot.
{
  int location = element.hash();
   HashTable[location] = new LLNode(element,HashTable[location] ;
  numElements++;
}

public static Hashable get(Hashable element)
   // Returns an element e from this list such that e.equals(element).
{
    LLNode temp = HashTable[element.Hash()];    //get head of LL of synonyms
    while(temp != null &amp;&amp; temp.obj.getKey().compareTo(key)!=0){
        temp = temp.next;
    }
    if(temp != null){   //return NULL or pointer to found node
    return temp.obj;
    }else{
    return null;
    }
}
```

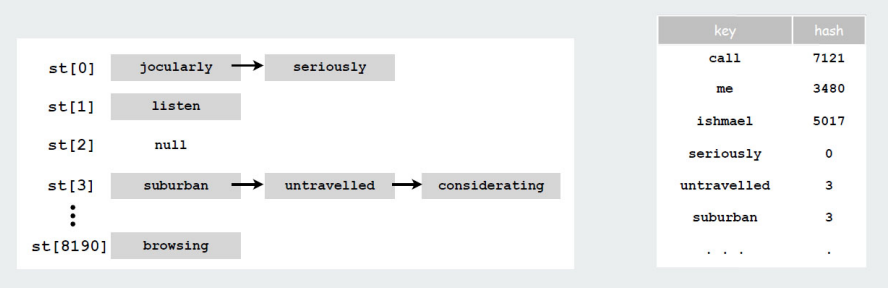
UPOTREBA ODVOJENOG ULANČAVANJA KOD MAPIRANJA STRINGOVA

Ukoliko je broj ključeva označen sa N, veličina heš tabele kod odvojenog ulančavanja se obično uzima kao M = N/2. Stoga će biti kreirano N/2 povezanih listi

Neka je broj ključeva označen sa *N*, i neka je veličina heš tabele koju ćemo koristiti označena sa *M*.

Pravilo koje najčešće važi kod odvojenog ulančavanja je da je veličina *M* približno jednaka *M = N/10*, tako da ćemo kreirati *M* povezanih listi.

Posmatrajmo sledeći primer kod koga se heš tabela sastoji iz 8190 polja, Slika-2.



Slika 4.3 Primer ulančanog heširanja za slučaj kada su ulazni ključevi stringovi: call, me, ishmael, seriously, untravelled i suburban.

Definišemo klasu *ListHashST*, koja će da bude naša heš lista, pri čemu klasa *Node* predstavlja čvor povezane liste. U okviru klase *ListHashST* definišemo funkcije za dodavanje i pristupanje poljima heš tabele:

```
public class ListHashST<Key, Value> {
 private int M = 8191;
 private Node[] st = new Node[M];

 private class Node {
  Object key;
  Object val;
  Node next;
  Node(Key key, Value val, Node next) {
   this.key = key;
   this.val = val;
   this.next = next;
  }
 }
 private int hash(Key key) {
  return (key.hashcode() &amp; 0x7fffffff) % M;
 }

 public void put(Key key, Value val) {
```

```
  int i = hash(key);
  for (Node x = st[i]; x != null; x = x.next)
   if (key.equals(x.key))  {
    x.val = val;
    return;
   }
  st[i] = new Node(key, value, first);
 }

 public Value get(Key key) {
  int i = hash(key);
  for (Node x = st[i]; x != null; x = x.next)
   if (key.equals(x.key))
    return (Value) x.val;
  return null;
 }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke:

Zadatak 1. Prikazati heš tabelu veličine 11 posle unosa elemenata sa ključevima 34, 29, 53, 44, 120, 39,45 i 40 upotrebom odvojenog ulančavanja.

Zadatak 2. Navesti osnovne razlike između linearne pretrage, kvadratne pretrage, dvostrukog heširanja i odvojenog ulančavanja.

Poglavlje 5

Faktor popunjenosti tabele i rehеširanje

FAKTOR POPUNJENOSTI TABELE

Faktor popunjenosti (load factor) predstavlja meru popunjenosti heš tabele. Poželjno je da faktor popunjenosti bude ispod 0.5 kod otvorenog adresiranja, a ispod 0.9 kod odvojenog ulanča

Faktor popunjenosti tabele (load factor- lambda) predstavlja meru popunjenosti heš tabele. Faktor popunjenosti λ heš tabele koja ima N elementa je dat jednačinom:

$\lambda = n / TableSize$

Kod linearne pretrage faktor popunjenosti heš tabele je u granicama $0 < \lambda < 1$ (donja granica može biti i veći od nule kod drugih metoda razrešavanja kolizije). Kod linearne pretrage važi pravilo da sa povećanjem broja *lamda* raste i broj kolizija.

Studije su pokazale da je poželjno održavati faktor popunjenosti heš tabele ispod 0.5 kod otvorenog adresiranja, a ispod 0.9 kod odvojenog ulančavanja.

Održavanje faktora popunjenosti ispod određene granice je bitno za poboljšanje performansi heširanja. U implementaciji klase *java.util.HashMap* usvojeno je da je prag za faktor popunjenosti **0.75.** Kada god faktor popunjenosti pređe ovaj prag, neophodno je uvećati veličinu heš tabele i izvršiti rehеširanje svih parova stare mape u novu veću mapu.

REHEŠIRANJE

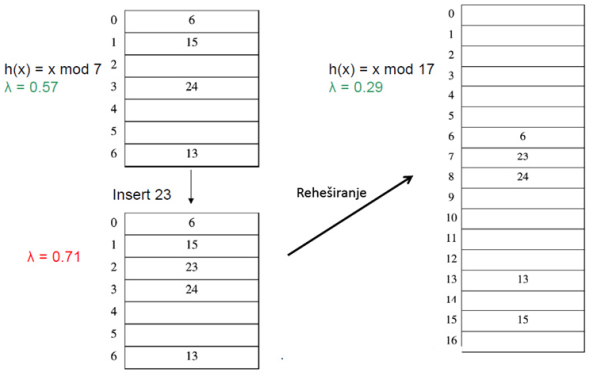
Kada god faktor popunjenosti tabele pređe odgovarajući prag, neophodno je uvećati veličinu heš tabele i izvršiti rehеširanje svih parova stare mape u novu veću mapu

Algoritam rehеširanja se sastoji iz sledećih koraka (Slika-1):

1. Alocirati novu heš tabelu koja je duplo veća od originalne tabele.

2. Prebaciti sve podatke iz originalne tabele u novu tabelu korišćenjem heš funkcije.

3. Uz pomoć odgovarajućeg koda, novu tabelu referencirati kao heš tabelu.



Slika 5.1 Primer rehеširanja

Rehеširanje treba izvršiti kada load factor pređe odgovarajuću toleranciju (npr. $\lambda \geq 0.5$), ili kada se posle određenog broja pokušaja ne uspe smestiti novi element u tabelu.

Napomena: Pri svakom rehеširanju menja se i heš funkcija, s obzirom da je izmenjena veličina heš tabele!

Da bi se smanjila verovatnoća rehеširanja, s obzirom da je to veoma zahtevna operacija, neophodno je barem duplirati veličinu heš tabele.

Čak i pri periodičnom rehеširanju, heširanje je i dalje jedan efikasan način implementacije mape.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke:

Zadatak 1. Ako heš tabela ima inicijalnu veličinu 4 i faktor popunjenosti je 0.5; prikazati tabelu posle ubacivanja elemenata sa ključevima 34, 29, 53, 44, 120, 39, 45 i 40 upotrebom linearne pretrage.

Zadatak 2. Ako heš tabela ima inicijalnu veličinu 4 i faktor popunjenosti je 0.5; prikazati tabelu posle ubacivanja elemenata sa ključevima 34, 29, 53, 44, 120, 39, 45 i 40 upotrebom kvadratne pretrage.

Zadatak 3. Ako heš tabela ima inicijalnu veličinu 4 i faktor popunjenosti je 0.5; prikazati tabelu posle ubacivanja elemenata sa ključevima 34, 29, 53, 44, 120, 39, 45 i 40 upotrebom odvojenog ulančavanja.

Poglavlje 6

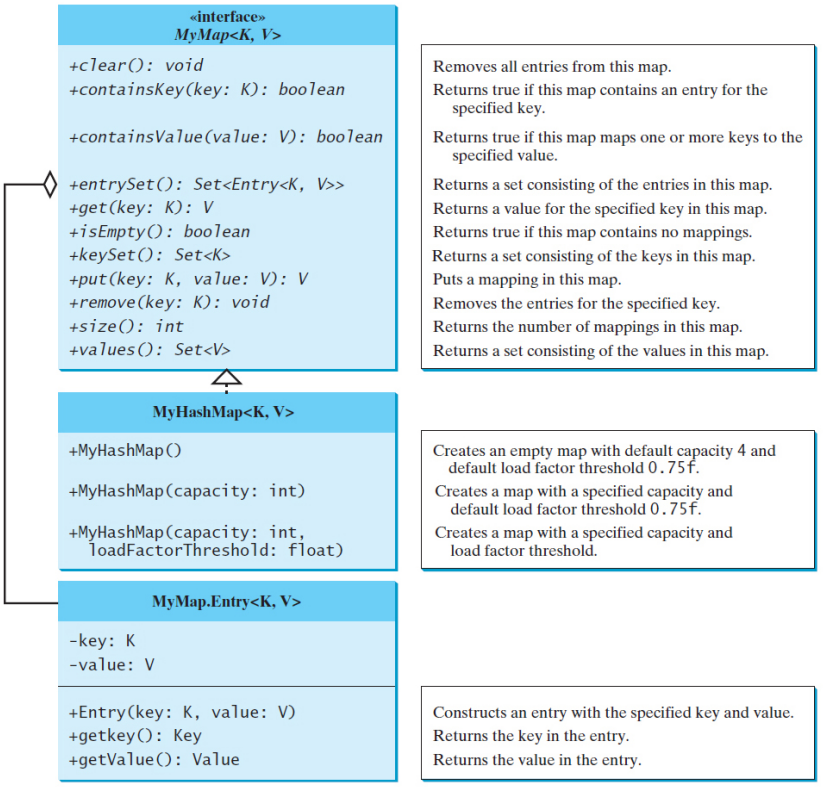
Implementacija Mape u Javi

IMPLEMENTACIJA MAPE KORIŠĆENJEM HEŠIRANJA

Klasa za mapu može biti implementirana na nekoliko različitih načina. Ukoliko implementiramo klasu HashMap korišćenjem heširanja, dostićićemo vreme pretraživanja složenosti O(1)

Do sada smo se upoznali sa konceptom heširanja, dizajniranjem dobre heš funkcije koja će da mapira ključ do indeksa u heš tabeli, merenjem performasni mape upotrebom faktora popunjenosti tabele, i načinom kako da uvećamo veličinu tabele reheširanje, da bi smo održali visoke performanse mapa. U nastavku lekcije ćemo se upoznati sa implementacijom mape korišćenjem razdvojenog ulančavanja.

Dizajniraćemo proizvoljan interfejs za mapu na isti način kako je kreiran standardan *java.util.Map*interfejsa, i kreiraćemo konkretnu klasu za mapu *MyHashMap*, koja odgovara standardnoj java klasi *HashMap*. Metode interfejsa *MyMap*, i klasa *MyHashMap* i podklase MyMap.Entry su prikazane na Slici 1.



Slika 6.1 Implementacija mape u Javi

INTERFEJS MYMAP

Intefejs Map definiše metode za pretraživanje, menjanje i dobijanje kolekcija vrednosti i seta ključeva

Kako da implementiramo klasu *MyHashMap*? Ukolko koristimo polje tipa *ArrayList* i ubacimo novi par na kraj liste, onda bi pretraga zahtevala vreme O(n). Ukoliko implementiramo klasu *MyHashMap* korišćenjem binarnog stabla, pretraga će zahtevati vreme $O(\log n)$, u slučaju da je stablo dobro izbalansirano. Na kraju, ukoliko implementiramo klasu *MyHashMap* korišćenjem heširanja, dostićićemo vreme pretraživanja složenosti $O(1)$. U narednom listingu je interfejs *MyMap*.

```
public interface MyMap<K, V> {
  /** Remove all of the entries from this map */
  public void clear();

  /** Return true if the specified key is in the map */
  public boolean containsKey(K key);

  /** Return true if this map contains the specified value */
  public boolean containsValue(V value);

  /** Return a set of entries in the map */
  public java.util.Set<Entry<K, V>> entrySet();

  /** Return the first value that matches the specified key */
  public V get(K key);

  /** Return true if this map contains no entries */
  public boolean isEmpty();

  /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet();

  /** Add an entry (key, value) into the map */
  public V put(K key, V value);

  /** Remove the entries for the specified key */
  public void remove(K key);

  /** Return the number of mappings in this map */
  public int size();

  /** Return a set consisting of the values in this map */
  public java.util.Set<V> values();

  /** Define inner class for Entry */
  public static class Entry<K, V> {
    K key;
```

```
    V value;

    public Entry(K key, V value) {
    this.key = key;
    this.value = value;
    }

    public K getKey() {
    return key;
    }

    public V getValue() {
    return value;
    }

    @Override
    public String toString() {
    return "[" + key + ", " + value + "]";
    }
  }
}
```

KLASA MYHASHMAP

Klasa MyHashMap implementira interfejs MyMap upotrebom odvojenog ulančavanja. Parametri na osnovu kojih se određuje veličina heš tabele i faktor popunjenosti su definisani u okviru klas

Klasa *MyHashMap* implementira interfejs *MyMap* upotrebom razdvojenog ulančavanja. Parametri na osnovu kojih se određuje veličina heš tabele i faktor popunjenosti su definisani u okviru klase. Podrazumevajuća vrednost inicijalanog kapaciteta heš tabele je 4, dok je maksimalni kapacitet 2 na 30. U prozivoljnom trenutku rada sa heš tabelom, vrednost kapaciteta je vrednost predstavljena kao 2 na N. Podrazumevajuća vrednosti faktora popunjenosti heš tabele je *0.75f*. Pri konstruisanju mape moguće je zadati proizvoljnu vrednost kada je u pitanju faktor popunjenosti. Proizvoljna tolerancija za faktor popunjenosti tabele je smeštena u polju *loadFactorThreshold*. Polje *size* označava trenutni broj ulaza (parova) u mapi. Heš tabela je predstavljena pomoću niza. Svaki element niza je ustvari povezana lista. Klasa *MyHashMap* ima tri različita konstruktora za kreiranje mape. Moguće je kreirati podrazumevajuću mapu, koja ima podrazumevajući kapacitet i faktor popunjenosti tabele, korišćenjem podrazumevajućeg konstruktora bez argumenata. Takođe je moguće, upotrebom konstruktora sa parametrima, kreirati heš tabelu sa zadatom vrednošću kapaciteta, i podrazumevajućom vrednošću faktora popunjenosti. Treći konstruktor se odnosi na kreiranja heš tabele korišćenjem zadatih vrednosti kapaciteta i faktora popunjenosti tabele.

```
import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {
```

```
// Define the default hash table size. Must be a power of 2
private static int DEFAULT_INITIAL_CAPACITY = 4;

// Define the maximum hash table size. 1 << 30 is same as 2^30
private static int MAXIMUM_CAPACITY = 1 << 30;

// Current hash table capacity. Capacity is a power of 2
private int capacity;

// Define default load factor
private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

// Specify a load factor used in the hash table
private float loadFactorThreshold;

// The number of entries in the map
private int size = 0;

// Hash table is an array with each cell that is a linked list
LinkedList<MyMap.Entry<K,V>>[] table;

/** Construct a map with the default capacity and load factor */
public MyHashMap() {
  this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a map with the specified initial capacity and
 * default load factor */
public MyHashMap(int initialCapacity) {
  this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a map with the specified initial capacity
 * and load factor */
public MyHashMap(int initialCapacity, float loadFactorThreshold) {
  if (initialCapacity > MAXIMUM_CAPACITY)
  this.capacity = MAXIMUM_CAPACITY;
  else
  this.capacity = trimToPowerOf2(initialCapacity);

  this.loadFactorThreshold = loadFactorThreshold;
  table = new LinkedList[capacity];
}

@Override /** Remove all of the entries from this map */
public void clear() {
  size = 0;
  removeEntries();
}

@Override /** Return true if the specified key is in the map */
public boolean containsKey(K key) {
  if (get(key) != null)
```

```
  return true;
  else
  return false;
}

@Override /** Return true if this map contains the value */
public boolean containsValue(V value) {
  for (int i = 0; i < capacity; i++) {
  if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
      if (entry.getValue().equals(value))
          return true;
  }
  }

  return false;
}

@Override /** Return a set of entries in the map */
public java.util.Set<MyMap.Entry<K,V>> entrySet() {
  java.util.Set<MyMap.Entry<K, V>> set =
  new java.util.HashSet<MyMap.Entry<K, V>>();

  for (int i = 0; i < capacity; i++) {
  if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
      set.add(entry);
  }
  }

  return set;
}

@Override /** Return the value that matches the specified key */
public V get(K key) {
  int bucketIndex = hash(key.hashCode());
  if (table[bucketIndex] != null) {
  LinkedList<Entry<K, V>> bucket = table[bucketIndex];
  for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key))
      return entry.getValue();
  }

  return null;
}

@Override /** Return true if this map contains no entries */
public boolean isEmpty() {
  return size == 0;
}
```

```
@Override /** Return a set consisting of the keys in this map */
public java.util.Set<K> keySet() {
  java.util.Set<K> set = new java.util.HashSet<K>();

  for (int i = 0; i < capacity; i++) {
  if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
      set.add(entry.getKey());
  }
  }

  return set;
}

@Override /** Add an entry (key, value) into the map */
public V put(K key, V value) {
  if (get(key) != null) { // The key is already in the map
  int bucketIndex = hash(key.hashCode());
  LinkedList<Entry<K, V>> bucket = table[bucketIndex];
  for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
      V oldValue = entry.getValue();
      // Replace old value with new value
      entry.value = value;
      // Return the old value for the key
      return oldValue;
      }
  }

  // Check load factor
  if (size >= capacity * loadFactorThreshold) {
  if (capacity == MAXIMUM_CAPACITY)
      throw new RuntimeException("Exceeding maximum capacity");

  rehash();
  }

  int bucketIndex = hash(key.hashCode());

  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] == null) {
  table[bucketIndex] = new LinkedList<Entry<K, V>>();
  }

  // Add a new entry (key, value) to hashTable[index]
  table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));

  size++; // Increase size

  return value;
}
```

```
@Override /** Remove the entries for the specified key */
public void remove(K key) {
  int bucketIndex = hash(key.hashCode());

  // Remove the first entry that matches the key from a bucket
  if (table[bucketIndex] != null) {
  LinkedList<Entry<K, V>> bucket = table[bucketIndex];
  for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
      bucket.remove(entry);
      size--; // Decrease size
      break; // Remove just one entry that matches the key
      }
  }
}

@Override /** Return the number of entries in this map */
public int size() {
  return size;
}

@Override /** Return a set consisting of the values in this map */
public java.util.Set<V> values() {
  java.util.Set<V> set = new java.util.HashSet<V>();

  for (int i = 0; i < capacity; i++) {
  if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
      set.add(entry.getValue());
  }
  }

  return set;
}

/** Hash function */
private int hash(int hashCode) {
  return supplementalHash(hashCode) &amp;amp; (capacity - 1);
}

/** Ensure the hashing is evenly distributed */
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
  capacity <<= 1;
  }
```

```
    return capacity;
  }

  /** Remove all entries from each bucket */
  private void removeEntries() {
    for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
        table[i].clear();
    }
    }
  }

  /** Rehash the map */
  private void rehash() {
    java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
    capacity <<= 1; // Double capacity
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Reset size to 0

    for (Entry<K, V> entry: set) {
    put(entry.getKey(), entry.getValue()); // Store to new table
    }
  }

  @Override
  public String toString() {
    StringBuilder builder = new StringBuilder("[");

    for (int i = 0; i < capacity; i++) {
    if (table[i] != null &amp;amp;&amp;amp; table[i].size() > 0)
        for (Entry<K, V> entry: table[i])
        builder.append(entry);
    }

    builder.append("]");
    return builder.toString();
  }
}
```

ANALIZA METODA KLASE MYHASHMAP

Vremenska složenost pojedinih metoda zavisi od parametra capacity. Stoga je neophodno pažljivo izabrati inicijalni kapacitet heš tabele kako bi se izbegle loše performanse ovih metoda

Metod *clear* briše parove iz mape. Iz njega se poziva metod *removeEntries()*, koji briše sve elemente povezane liste koja pripada tom polju mape. Metod *containsKey(key)* proverava da li se određeni ključ nalazi u mapi, tako što poziva metod *get*. Metod *containsValue(value)*proverava da li se vrednost *value* nalazi u mapi. Vreme izvršavanja ovog

metoda je *O*(*capacity*), pošto metod *entrySet()* kao rezultat vraća set koji sadrži sve parove u mapi. Metod *keySet()* vraća set koji sadrži sve ključeve u mapi. Metod *values()* vraća sve vrednosti iz mape.

Na narednoj Slici 2 je sumirana vremenska složenost odgovarajućih metoda klase *MyHashMap* (pa i *HashMap*).

Metod	*Vreme*
`clear()`	$O(capacity)$
`containsKey(key: Key)`	$O(1)$
`containsValue(value: V)`	$O(capacity)$
`entrySet()`	$O(capacity)$
`get(key: K)`	$O(1)$
`isEmpty()`	$O(1)$
`keySet()`	$O(capacity)$
`put(key: K, value: V)`	$O(1)$
`remove(key: K)`	$O(1)$
`size()`	$O(1)$
`values()`	$O(capacity)$
`rehash()`	$O(capacity)$

Slika 6.2 Analiza vremenske složenosti metoda

Metod *put(key, value)* dodaje nov par u mapu. Metod prvo proverava da li se ključ nalazi u mapi. Ukoliko je odgovor da, on ga locira u mapi i staru vrednost zamenjuje sa novom vrednošću, dok kao rezultat ovaj metod vraća staru vrednost. Ukoliko je ključ nov u mapi, kreira se nov ulaz u mapu. Pre ubacivanja novog ulaza u mapu, metod proverava da li bi se ubacivanjem novog ulaza probila tolerancija za faktor popunjenosti mape. Ukoliko je odgovor da, poziva se metod *rehash()* kako bi se uvećao kapacitet tabele, a zatim se par ubacuje u novu tabelu.

Metod *rehash()* prvo kopira sve ulaze mape u set, zatim udvostručava kapacitet, kreira novu inicijalnu heš tabelu mape, i zatim kopira vrednosti iz seta u novu heš tabelu.

S obzirom da se rehеširanje ne dešava dosta često, vremenska složenost metoda *put* može da se kađe da je *O*(1). Napomenimo da vremenska složenost metoda *clear*, *entrySet*, *keySet*, **values**, i **rehash** zavisi od parametra *capacity*, stoga je neophodno pažljivo izabrati inicijalni kapacitet heš tabele kako bi se izbegle loše performanse ovih metoda.

UPOTREBA KLASE MYHASHMAP

Klasa HashMap je efikasna za lociranje neke vrednosti, ubacivanje ulaza i za brisanje ulaza. Nju koristimo kada nam nije bitan redosled ubacivanja parova niti uređenost parova u mapi

Za testiranje implementirane mape možemo da koristimo sledeću test klasu:

```
public class TestMyHashMap {
  public static void main(String[] args) {
    // Create a map
```

```
    MyMap<String, Integer> map = new MyHashMap<String, Integer>();
    map.put("Smith", 30);
    map.put("Anderson", 31);
    map.put("Lewis", 29);
    map.put("Cook", 29);
    map.put("Smith", 65);

    System.out.println("Entries in map: " + map);

    System.out.println("The age for " + "Lewis is " +
    map.get("Lewis"));

    System.out.println("Is Smith in the map? " +
    map.containsKey("Smith"));
    System.out.println("Is age 33 in the map? " +
    map.containsValue(33));

    map.remove("Smith");
    System.out.println("Entries in map: " + map);

    map.clear();
    System.out.println("Entries in map: " + map);
  }
}
```

Rezultat:

```
Entries in map: [[Anderson, 31][Smith, 65][Lewis, 29][Cook, 29]]
The age for Lewis is 29
Is Smith in the map? true
Is age 33 in the map? false
Entries in map: [[Anderson, 31][Lewis, 29][Cook, 29]]
Entries in map: []
```

Program kreira mapu korišćenjem klase *MyHashMap* i dodaje 5 ulaza u mapu. Mapa ustvari sadrži 4 vrednosti jer dva para imaju isti ključ (Smit).

U liniji 5 se dodaje ključ Smith vrednosti 30 a u liniji 9 se dodaje ključ Smith vednosti 65. Poslednja vrednosti zamenjuje prethodnu. Mapa ustvari ima samo četiri ulaza. Program prikazuje parove mape (linija 11), za svaki ključ dobija vrednost (linija 14), proverava da li mapa sadrži ključ (linija 17) i vrednost (linija 19), briše iz mape podatak čiji je ključ Smith (linija 21), i ponovo prikazuje sve parove mape (linija 22). Konačno, program briše mapu (linija 24) i prikazuje praznu mapu (linija 25).

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Šta znači 1 << 30? Koji celi brojevi se dobijaju operacijama 1 << 1,1 << 2, and 1 << 3?

Zadatak 2. Koji celi brojevi se dobijaju sledećim operacijama 32 >> 1, 32 >> 2, 32 >> 3 i 32 >> 4?

Zadatak 3. U implementaciji *MyHashMap* da li će program da radi ako se *LinkedList* zameni sa *ArrayList*?

Zadatak 4. Opisati kako je metod **put(key, value)** implementiran u *MyHashMap* klasi.

Zadatak 5. Koji je izlaz sledećeg koda?

```
MyMap<String, String> map = new MyHashMap<String, String>();
map.put("Texas", "Dallas");

map.put("Oklahoma", "Norman");
map.put("Texas", "Austin");
map.put("Oklahoma", "Tulsa");

System.out.println(map.get("Texas"));
System.out.println(map.size());
```

Zadatak 6. Napisati JavaFX aplikaciju koja animira rad *MyHashMap*, kao što je prikazano na slici **Odvojeno ulančavanje-Slika2**. Možete promeniti početnu veličinu tabele. Podrazumeva se da je granična vrednost faktora popunjenosti 0.75.

Zadatak 7. Kreirati novu konkretnu klasu koja implementira interfejs *MyMap* korišćenjem linearne probe. Radi uprošćenja, koristiti *f(key) = key%size* kao heš funkciju, gde je *size* veličina heš tabele. Početna veličina heš tabele je **4**. Veličina heš tabele se duplira svaki put kada faktor popunjenosti pređe prag od 0.5.

Zadatak 8. Kreirati novu konkretnu klasu koja implementira interfejs *MyMap* korišćenjem kvadratne probe. Radi uprošćenja, koristiti *f(key) = key % size* kao heš funkciju, gde je *size* veličina heš tabele. Inicijalno, veličina heš tabele **4**. Veličina heš tabele se duplira svaki put kada faktor popunjenosti pređe prag od 0.75.

Zadatak 9. Kreirati novu konkretnu klasu koja implementira interfejs *MyMap* korišćenjem dvostrukog heširanja. Koristiti *f(key) = key % size* kao heš funkciju, gde je *size* veličina heš tabele. Inicijalno, veličina heš tabele je **8**. Veličina heš tabele se duplira svaki put kada faktor popunjenosti pređe prag od 0.65.

Zadatak 10. Modifikovati klasu *MyHashMap* tako da se dozvole duplikati ključeva u ulazima mape. Neophodno je modifikovati implementaciju za metod *put(key, value)*. Takođe kreirati novi metod pod nazivom *getAll(key)*koji vraća skup vrednosti koji imaju isti ključ u mapi.

Poglavlje 7

Implementacija seta u Javi

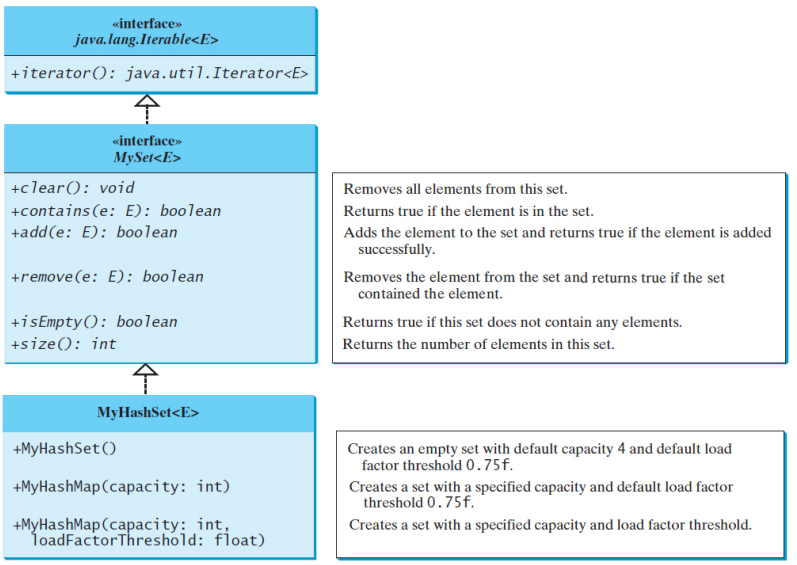
IMPLEMENTACIJA SETA KORIŠĆENJEM HEŠIRANJA

Postoje tri konkretne implementacije seta u Javi i to java.util.HashSet, java.util.LinkedHashSet, i java.util.TreeSet. Klasa java.util.HashSet je implementirana korišćenjem heširanja

Skup (set) je struktura podataka koja predstavlja kolekciju bez duplikata. U JCF-u je definisan interfejs *java.util.Set* za rad sa skupovima. Postoje tri konkretne implementacije seta u Javi i to *java.util.HashSet*, *java.util.LinkedHashSet*, i *java.util.TreeSet*. Klasa *java.util.HashSet*je implementirana korišćenjem heširanja, *java.util.LinkedHashSet* korišćenjem povezane liste, tj. klase *LinkedList*, dok je klasa *java.util.TreeSet*implementirana korišćenjem crveno-crnih stabala o čemu će biti više reči u lekciji o stablima i pretraživanju.

U nastavku je dat primer implementacije klase *MyHashSet* koja je izvršena na isti način kao i implementacija klase *MyHashMap*. Jedina razlika je u tome što su u mapi sačuvani parovi ključ/vrednost dok je u skupu sačuvana kolekcija elemenata. U nastavku ćemo kreirati proizvoljan interfejs *MySet* na isti način kako je u Javi implementiran interfejs *java.util.Set* i kreiraćemo klasu *MyHashSet*, na isti način kako je u Javi implementirana klasa *MyHashSet*. Spisak metoda i hijerarhija klasa i interfejsa je prikazana na sledećoj Slici 1.

Implementacija klase *MyHashSet* je veoma slična implementaciji klase *MyHashMap* osim sledećih razlika:



Slika 7.1 Klasa MyHashSet implementira intefejs MySet

1. U heš tabeli za *MyHashSet* se čuvaju elementi, dok se u heš tabeli za *MyHashMap* čuvaju parovi (ključ/vrednost).

2. *MySet* proširuje *java.lang.Iterable* a *MyHashSet* implementira *MySet* i predefiniše metod *iterator()*. Stoga je moguće koristiti iterator pri radu sa objektima tipa *MyHashSet*.

INTERFEJS MYSET I KLASA MYHASHSET

MySet proširuje java.lang.Iterable a MyHashSet implementira MySet i predefiniše metod iterator(). Stoga je moguće koristiti iterator pri radu sa objektima tipa MyHashSet

Naredni listing prikazuje interfejs *MySet*, koji proširuje interfejs java.lang.Iterable za podršku iteratoru.

```
public interface MySet<E> extends java.lang.Iterable<E> {
  /** Remove all elements from this set */
  public void clear();

  /** Return true if the element is in the set */
  public boolean contains(E e);

  /** Add an element to the set */
  public boolean add(E e);

  /** Remove the element from the set */
  public boolean remove(E e);

  /** Return true if the set contains no elements */
  public boolean isEmpty();

  /** Return the number of elements in the set */
  public int size();

  /** Return an iterator for the elements in this set */
  public java.util.Iterator iterator();
}
```

U nastavku je primer implementacije klase *MyHashSet* korišćenjem razdvojenog ulančavanja.

```
import java.util.LinkedList;

public class MyHashSet<E> implements MySet<E> {
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
```

```
private int capacity;

// Define default load factor
private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

// Specify a load factor threshold used in the hash table
private float loadFactorThreshold;

// The number of elements in the set
private int size = 0;

// Hash table is an array with each cell that is a linked list
private LinkedList<E>[] table;

/** Construct a set with the default capacity and load factor */
public MyHashSet() {
  this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a set with the specified initial capacity and
 * default load factor */
public MyHashSet(int initialCapacity) {
  this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a set with the specified initial capacity
 * and load factor */
public MyHashSet(int initialCapacity, float loadFactorThreshold) {
  if (initialCapacity > MAXIMUM_CAPACITY)
  this.capacity = MAXIMUM_CAPACITY;
  else
  this.capacity = trimToPowerOf2(initialCapacity);

  this.loadFactorThreshold = loadFactorThreshold;
  table = new LinkedList[capacity];
}

@Override /** Remove all elements from this set */
public void clear() {
  size = 0;
  removeElements();
}

@Override /** Return true if the element is in the set */
public boolean contains(E e) {
  int bucketIndex = hash(e.hashCode());
  if (table[bucketIndex] != null) {
  LinkedList<E> bucket = table[bucketIndex];
  for (E element: bucket)
      if (element.equals(e))
      return true;
  }
```

```
  return false;
}

@Override /** Add an element to the set */
public boolean add(E e) {
  if (contains(e)) // Duplicate element not stored
  return false;

  if (size > capacity * loadFactorThreshold) {
  if (capacity == MAXIMUM_CAPACITY)
      throw new RuntimeException("Exceeding maximum capacity");

  rehash();
  }

  int bucketIndex = hash(e.hashCode());

  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] == null) {
  table[bucketIndex] = new LinkedList<E>();
  }

  // Add e to hashTable[index]
  table[bucketIndex].add(e);

  size++; // Increase size

  return true;
}

@Override /** Remove the element from the set */
public boolean remove(E e) {
  if (!contains(e))
  return false;

  int bucketIndex = hash(e.hashCode());

  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] != null) {
  LinkedList<E> bucket = table[bucketIndex];
  for (E element: bucket)
      if (e.equals(element)) {
      bucket.remove(element);
      break;
      }
  }

  size--; // Decrease size

  return true;
}

@Override /** Return true if the set contains no elements */
```

```
public boolean isEmpty() {
  return size == 0;
}

@Override /** Return the number of elements in the set */
public int size() {
  return size;
}

@Override /** Return an iterator for the elements in this set */
public java.util.Iterator<E> iterator() {
  return new MyHashSetIterator(this);
}

/** Inner class for iterator */
private class MyHashSetIterator implements java.util.Iterator<E> {
  // Store the elements in a list
  private java.util.ArrayList<E> list;
  private int current = 0; // Point to the current element in list
  private MyHashSet<E> set;

  /** Create a list from the set */
  public MyHashSetIterator(MyHashSet<E> set) {
  this.set = set;
  list = setToList();
  }

  @Override /** Next element for traversing? */
  public boolean hasNext() {
  if (current < list.size())
      return true;

  return false;
  }

  @Override /** Get current element and move cursor to the next */
  public E next() {
  return list.get(current++);
  }

  @Override /** Remove the current element and refresh the list */
  public void remove() {
  // Delete the current element from the hash set
  set.remove(list.get(current));
  list.remove(current); // Remove current element from the list
  }
}

/** Hash function */
private int hash(int hashCode) {
  return supplementalHash(hashCode) &amp;amp;amp; (capacity - 1);
}
```

```
/** Ensure the hashing is evenly distributed */
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
  capacity <<= 1;
  }

  return capacity;
}

/** Remove all e from each bucket */
private void removeElements() {
  for (int i = 0; i < capacity; i++) {
  if (table[i] != null) {
      table[i].clear();
  }
  }
}

/** Rehash the set */
private void rehash() {
  java.util.ArrayList<E> list = setToList(); // Copy to a list
  capacity <<= 1; // Double capacity
  table = new LinkedList[capacity]; // Create a new hash table
  size = 0; // Reset size

  for (E element: list) {
  add(element); // Add from the old table to the new table
  }
}

/** Copy elements in the hash set to an array list */
private java.util.ArrayList<E> setToList() {
  java.util.ArrayList<E> list = new java.util.ArrayList<E>();

  for (int i = 0; i < capacity; i++) {
  if (table[i] != null) {
      for (E e: table[i]) {
      list.add(e);
      }
  }
  }

  return list;
}

@Override
```

```
  public String toString() {
    java.util.ArrayList<E> list = setToList();
    StringBuilder builder = new StringBuilder("[");

    // Add the elements except the last one to the string builder
    for (int i = 0; i < list.size() - 1; i++) {
    builder.append(list.get(i) + ", ");
    }

    // Add the last element in the list to the string builder
    if (list.size() == 0)
    builder.append("]");
    else
    builder.append(list.get(list.size() - 1) + "]");

    return builder.toString();
  }
}
```

ANALIZA METODA KLASE MYHASHSET

Kao i pri radu sa mapama, kod setova je takođe neophodno pažljivo izabrati inicijalni kapacitet heš tabele kako bi se izbegle loše performanse pojedinih metoda

Kao i kod klase *MyHashMap* imamo tri konstruktora koji rade po istom principu.

Metod *clear* briše elemente iz seta. On poziva metod *removeElements()*, koji briše celu tabelu. Svaka ćelija heš tabele je povezana lista u kojoj se čuvaju elementi koji imaju isti heš kod. Metod *contains(element)* proverava da lise određeni element nalazi u skupu, ispitujući svaku povezanu listu heš tabele posebno. Metod *add(element)* dodaje novi element u skup. Ovaj metod prvo proverava da li se element već nalazi u skupu. Ukoliko je odgovor da, meto vraća false kao rezultat. Metod zatim proverava da li veličina tabele probija vrednost faktora popunjenosti tabele. U tom slučaju program poziva metod *rehash()* u cilju povećanja kappaciteta i kreiranja nove veće heš tabele za čuvanje elemenata.

Metod *rehash()* prvo kopira elemente u listu, duplira kapacitet tabele, postavlja novu vrednost praga faktora popunjenosti, kreira novu tabelu i inicijalizuje njenu veličinu na 0. Metod zatim kopira elemente u veću heš tabelu. Ovaj metod zahteva $O(capacity)$ vreme. Ukoliko se ne vrši reheširanje, metod *add* je složenosti $O(1)$. Metod *iterator()* vraća instancu iteratora tipa *java.util.Iterator*. Na narednoj Slici 2 je sumirana vremenska složenost odgovarajućih metoda klase *MyHashSet*.

Klasa *MyHashSetIterator*implementira interfejs *java.util.Iterator*u cilju kreiranja iteratora za kretanje unapred. Kada je *MyHashSetIterator* konstruisan, on kopira sve elemente is skupa u listu. Promenljiva *current* pokazuje na element u listi. Inicijalno *current* je **0** što znači da pokazuje na prvi element u listi. *MyHashSetIterator* implementira metode *hasNext()*, *next()*, i *remove()* iz *java.util.Iterator*. Pozivanjem metode *hasNext()* dobija se rezultat ***true*** ako važi da je *current < list.size()*. Metod *next()* vraća trenutni element i pomera *current* da pokazuje

na sledeći element. Pozivanje metoda *remove()* briše se trenutni element koji je u itaratoru iz seta.

Metod	*Vreme*
`clear()`	$O(capacity)$
`contains(e: E)`	$O(1)$
`add(e: E)`	$O(1)$
`remove(e: E)`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`iterator()`	$O(capacity)$
`rehash()`	$O(capacity)$

Slika 7.2 Vremenska složenost odgovarajućih metoda klase MyHashSet

PRIMER UPOTREBE KREIRANE KLASE ZA SET

Klasa HashSet se koristi za skladištenje elemenata bez dupliranja, tj. koji se ne ponavljaju. Svaki pokušaj dodavanja već postojećeg elementa u skup će biti zanemaren

Naredni listing predstavlja primer upotrebe i testiranja klase *MyHashSet*.

```
public class TestMyHashSet {
  public static void main(String[] args) {
    // Create a MyHashSet
    MySet<String> set = new MyHashSet<String>();
    set.add("Smith");
    set.add("Anderson");
    set.add("Lewis");
    set.add("Cook");
    set.add("Smith");

    System.out.println("Elements in set: " + set);
    System.out.println("Number of elements in set: " + set.size());
    System.out.println("Is Smith in set? " + set.contains("Smith"));

    set.remove("Smith");
    System.out.print("Names in set in uppercase are ");
    for (String s: set)
    System.out.print(s.toUpperCase() + " ");

    set.clear();
    System.out.println("\nElements in set: " + set);
  }
}
```

Rezultat programa:

```
Elements in set: [Cook, Anderson, Smith, Lewis]
Number of elements in set: 4
Is Smith in set? true
Names in set in uppercase are COOK ANDERSON LEWIS
Elements in set: []
```

Prethodni primer kreira skup korišćenjem klase *MyHashSet* i dodaje 5 elemenata u njega. Kao i u primeru sa mapama, dva puta se dodaje objekat pod nazivom Smith, pa će stoga biti samo jedan od njih u skupu, pošto skup ne dozvoljava ubacivanje i skladištenje duplikata. Kreirani skup ustvari sadrži 4 elementa.

ZADACI ZA SAMOSTALNI RAD

Zadaci za samostalni rad

Zadatak 1. Zašto je moguće koristiti **for-each** petlju za obilazak elemenata u heš skupu?

Zadatak 2. Opisati kako je metod **add(e)** implementiran u *MyHashSet* klasi.

Zadatak 3. U implementaciji *MyHashSet*, metod **remove** briše trenutni element iteratora. Pored toga se element briše i iz interne liste (line 161):

```
list.remove(current); // Remove current element from the list
```

Da li je i to potrebno? Zašto?

Zadatak 4. Implementirati klasu *MyHashSet* korišćenjem klase *MyHashMap*. Treba napomenuti da je moguće kreirati uređeni par *(key, key)*, umesto *(key, value)*.

Zadatak 5. (*Uporediti MyHashSet i MyArrayList*) *MyArrayList* je definisana u lekciji o listama. Napisati program koji generiše **1000000** random celih brojeva između **0** i **999999**, meša ih (shuffle), a zatim ih pakuje u *MyArrayList* i u *MyHashSet*. Generisati listu od **1000000** random celih brojeva između **0** i **1999999**. Za svaki broj u listi, testirati da li je u listi baziranoj na nizu ili u heš setu. Pokrenuti program i prikazati ukupno vreme testiranja za niz i za heš set.

Zadatak 6. Napisati JavaFX aplikaciju koja animira linearnu pretragu, kao što je prikazano na **Linearna pretraga-Slika2**. Možete promeniti početnu veličinu tabele. Podrazumeva se da je granična vrednost faktora popunjenosti 0.75.

Zadatak 7. Napisati JavaFX aplikaciju koja animira kvadratnu pretragu, kao što je prikazano na **Kvadratna pretraga-Slika4**. Možete promeniti početnu veličinu tabele. Podrazumeva se da je granična vrednost faktora popunjenosti 0.75.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Cilj heširanja je da se ubrza pretraživanje podataka u nekom skupu ili mapi, bez obilaska svih elemenata odnosno podataka u skupu. Heš tabela (engl. hash table) ili heš mapa (engl. hash map) je struktura podataka koja koristi heš funkciju za efikasno preslikavanje određenih ključeva u njima pridružene vrednosti.

Heš funkcija je svaki algoritam koji podacima proizvoljne dužine dodeljuje podatke fiksne dužine. Vrednost koju vraća heš funkcija zove se heš vrednost ili heš kod. Tipična heš funkcija prvo konvertuje ključ pretrage u neku celobrojnu vrednost koja se naziva heš kod, a zatim se vrši kompresija heš koda u indeks koji odgovara polju heš tabele.

Svaka klasa u programskom jeziku Java ekspicitno ili implicitno sadrži metodu *hashCode()*, koja pretvara podatak instance klase u jedinstvenu heš vrednost. Heš kodovi primitivnih tipova se kreiraju korišćenjem univerzalnog heširanja gde se slučajno bira funkcija iz familije heš funkcija sa ciljem da se ostvari što manji broj sudara ključeva. Heš kod nekog ključa može biti ogroman ceo broj tako da on mora biti skaliran kako bi mogao da bude u opsegu indeksa heš tabele. Skaliranje heš koda se uglavnom vrši korišćenjem funkcije moduo (ostatak pri deljenju), pri čemu se određuje ostatak pri deljenju ključa sa vrednošću koja predstavlja veličinu tabele.

Do sudara ili kolizije ključeva dolazi kada heš funkcija preslika dva različita ključa u istu adresu u heš tabeli. Pogrešno implementirana metoda *hashCode()* korisnički definisane klase je veoma čest uzrok pojave sudara ključeva, pri mapiranju podataka u heš mapu. Postoje dva pristupa u razrešavanju kolizija: odvojeno ulančavanje (korišćenjem povezanih listi za svako polje u tabeli) i otvoreno adresiranje (korišćenjem nizova za tabelu).

Ideja razrešavanja kolizije kod otvorenog adresiranja je da se podatak koji je u koliziji smesti na neko drugo mesto u tabeli koje nije zauzeto. Metode otvorenog adresiranja su: Linearna pretraga, Kvadratna pretraga, Dvostruko heširanje.

Prema metodi odvojenog ulančavanja, svaka pozicija u heš tabeli sadrži listu neograničene dužine u koju će biti smešteni podaci sa istim heš kodom. Kolekcija na odgovarajućem polju heš tabele se najčešće formira korišćenjem povezane liste koje ustvari ima oblik lanca, pa se stoga i metoda naziva ulančavanje.

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, deseto izdanje, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.

[3] http://ydtech.blogspot.rs/2010/06/hashmap-hashcode-collision-by-example.html

[4] https://en.wikipedia.org/wiki/Hash_table#Collision_resolution

[5] https://en.wikipedia.org/wiki/Open_addressing

[6] http://webdocs.cs.ualberta.ca/~holte/T26/open-addr.html

[7] https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm