Hash tablice

Slavić, Duje

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Duje Slavić

Hash tablice

Završni rad

Osijek, 2019.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Duje Slavić

Hash tablice

Završni rad

Voditelj: doc. dr. sc. Slobodan Jelić

Osijek, 2019.

Sažetak

U ovom završnom radu ćemo se upoznati s hash tablicama i hash funkcijama. Proučiti ćemo neka rješenja problema koji se javljaju prilikom korištenja hash tablica, kao i neke metode modeliranja hash funkcija. Na kraju ćemo na jednostavnom primjeru demostrirati korištenje hash tablice u pythonu.

Ključne riječi: hash, hashiranje

Abstract

In this final paper we will introduce hash tables and hash functions. We will also examine some of the solutions of the problems raised by the usage of hash tables, as well as some of the methods of designing hash functions. In the end we will look at a simple example using python, made to demonstrate the usage of hash functions.

Key words: hash, hashtable, hashing, probe

Sadržaj

1 Uvod

Glazba, jezik duše. Gotovo da ne postoji osoba koja ju ne sluša, barem povremeno, što uopće nije iznenađujuće s obzirom na njezinu raznolikost i ogroman katalog. Konstantno uživamo u otkrivanju novih pjesama.

Koliko Vam se puta dogodilo da ste u kafiću, klubu, restoranu ili na plaži pored nekog beach bara i čujete nepoznatu pjesmu koja Vas mami melodijom, ali nitko oko Vas ne zna ni ime pjesme, ni ime izvođača. Pjesma završi. Tko zna kada ćete ju opet čuti? Hoćete li ju opet čuti?

Je li Vam se ikada dogodilo da čujete neku poznatu pjesmu, ali ne možete dokučiti njeno ime, usprkos silnom napinjanju svake moždane stanice u svrhu pretraživanja pamćenja? Pjesma završi. Tko zna kada ćete se sjetiti imena? Hoćete li se sjetiti?

Ljudi koji su napravili aplikaciju Shazam su riješili naša oba problema. Ukratko, dok pjesma svira, aplikacija će preko Vašeg uređaja (mobitela, tableta, ...) čuti pjesmu i vama pokazati ime izvođača i naziv pjesme. Ako ste koristili Shazam, znate da aplikacija, osim zanimljive ideje, ima i zapanjujuću brzinu pronalaska pjesme.

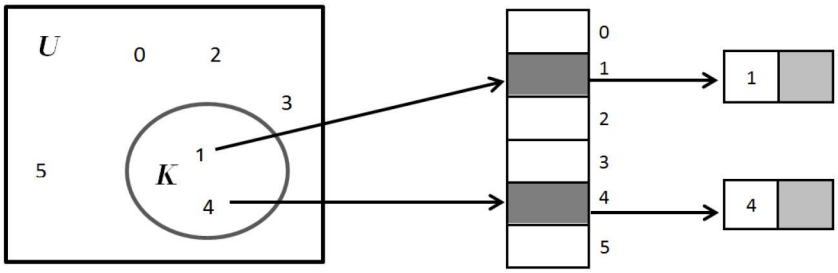
Kako aplikacija funkcionira?

Očito su morali spremiti informacije o svim tim pjesmama. Ali, pitanje je kako doći od snimke dijela pjesme do same pjesme (u bazi)? Nema smisla pretraživati čitavu bazu pjesama i uspoređivati sa zapisanim zvukom.

Pojednostavljeno, njihov algoritam stvara karakterističnu 'otisak' pjesme preko vršnih frekvencija te se preko tog otiska pronalazi pjesma u njihovoj bazi. Pjesme su spremljenu u hash tablicu preko koje se jednostavno može pronaći traženi element jer svaki element hash tablice je povezan sa svojim ključem preko hash funkcije. Ta funkcija će, u odnosu na ključ, pohraniti podatke na točno određeno mjesto u tablici, stoga s ključem možemo brzo doći do podataka.

2 Direktno adresiranje

Označimo s U skup svih ključeva. Koristeći polje $T[0..m-1]$, prikažimo dinamičan skup u kojemu svaki element ima ključ koji pripada skupu $U = \{0, 1, \ldots, m-1\}$, gdje m nije prevelik i u kojemu nikoja dva elementa nemaju isti ključ. U tom polju, svaka pozicija odgovara pripadnom ključu iz U. Pozicija k pokazuje na element u skupu čiji je ključ k. Ukoliko skup ne sadrži element s ključem k, tada je $T[k]$ =NIL. Pogledaj sliku 2.1.



Slika 2.1: Direktno adresiranje

Implementacija potrebnih funkcija, prikazana je algoritmima 1, 2 i 3.

Algoritam 1

function DIRECT-ADDRESS-SEARCH(T, k)
  return $T[k]$

Algoritam 2

function DIRECT-ADDRESS-INSERT(T, x)
  $T[key[x]] \leftarrow x$

Algoritam 3

function DIRECT-ADDRESS-DELETE(T, x)
  return $T[key[x]] \leftarrow NIL$

Vrijeme potrebno za izvršavanje svake od gore navedenih operacija je $O(1)$.

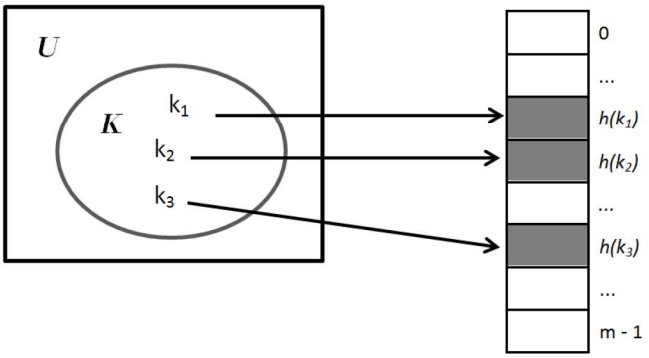
3 Hash tablice

Problem s direktnim adresiranjem se javlja kada je skup U velik pa spremanje tablice T veličine $|U|$ može biti nepraktično. Uostalom, skup K, koji se sastoji od ključeva koji su uistinu pohranjeni, može biti znatno manji u odnosu na U, stoga bi većina memorijskog prostora rezerviranog za U bila uzaludna.

Kada je skup K znatno manji od U, hash tablica zahtjeva manje memorije od direktno adresirajuće tablice. Preciznije rečeno, memorijske potrebe se mogu smanjiti na $\Theta(|K|)$, uz zadržavanje prednosti da je vrijeme potrebno za traženje elementa hash tablice $O(1)$.

Kod direktnog adresiranja, element s ključem k je pohranjen na poziciju k, dok je pri korištenju hash tablice takav element pohranjen na poziciju $h(k)$, gdje je h neka hash funkcija koja računa poziciju ključa k. Ovdje h povezuje skup U s pozicijama hash tablice $T[0..m-1]$, tj. $h : U \to \{0, 1, \ldots, m-1\}$. Vidi sliku 3.1.

Kažemo da je $h(k)$ hash vrijednost ključa k. Svrha hash funkcije je smanjiti raspon indeksa reda kojeg rabimo. Umjesto $|U|$ vrijednosti,trebamo samo s njih m, čime su memorijske potrebe smanjene.

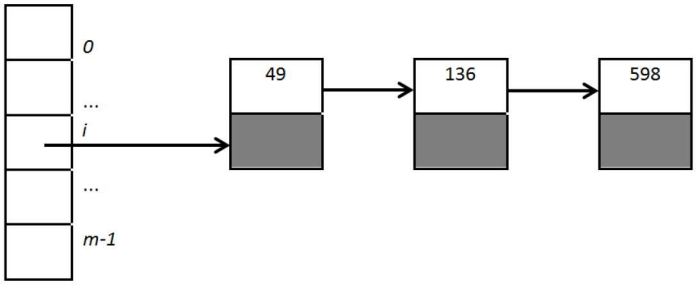


Slika 3.1: Hash tablica

Moguće je da dva ključa pripadnu istoj poziciji, takozvani "sudar". Idealno rješenje ovog problema bi bilo da se izbjegne nastanak sudara, to možemo pokušati postići odabirom pogodne funkcije h. Jedna ideja je pokušati napraviti h tako da se doima nasumičnim, čime bi se izbjegli sudari ili pak smanjio njihov broj. Pošto je $|U| > m$, postoje barem dva ključa s istom hash vrijednošću, stoga je izbjegavanje sudara, u cijelosti, nemoguće. No, dok dobro osmišljena "nasumična" hash funkcija minimizira pojavu sudara, i dalje nam je potrebna metoda kojom ćemo razriješiti sudare koji se pojave.

3.1 Ulančavanje

Kod ulančavanja, sve elemente koji se vežu na istu poziciju stavljamo u povezanu listu (Slika 3.2). Pozicija i sadrži pokazivač na glavu liste svih elemenata pridruženih i. Ukoliko takvi elementi ne postoje, pozicija i sadrži NIL.



Slika 3.2: Ulančavanje

Operacije rječnika nad hash tablicom T, kada su sudari razriješeni ulančavanjem, definiramo preko algoritama 4, 5 i 6.

Algoritam 4

function CHAINED-HASH-INSERT(T, x)
  insert x **at the head of list** $T[h(key[x])]$

Algoritam 5

function CHAINED-HASH-SEARCH(T, k)
  search for an element with key k **in list** $T[h(k)]$

Algoritam 6

function CHAINED-HASH-DELETE(T, x)
  delete x **from the list** $T[h(key[x])]$

Pretpostavimo da neki element x, kojeg unosimo u tablicu, ne postoji u tablici T. Njegov unos će nas Koštati $O(1)$ jer znamo točno na koje mjesto će ga hash funkcija spremiti, a ukoliko na toj poziciji imamo listu elemenata, ne moramo prolaziti listom jer ga stavljamo na početak liste. Najgori slučaj, prilikom traženja, bi bio da se svaki element naše tablice T spremio na istu poziciju. To bi značilo, ukoliko želimo pronaći neki element moramo pretraživati listu, a u najgorem slučaju moramo proći čitavu listu duljine $|S|$.

Najgori vremenski slučaj, za unos, je $O(1)$ jer se pretpostavlja da element x, kojeg unosimo, nije postojan u tablici T. Za traženje, najgori vremenski slučaj je proporcionalan duljini liste. Brisanje elementa x se može postići u vremenu $O(1)$, ukoliko su liste dvostruko povezane.

3.2 Analiza korištenja ulančavanja

Uzmimo hash tablicu T s m pozicija koja sadrži n elemenata. Faktor opterećenja α definiramo kao n/m, tj. srednji broj elemenata spremljenih u lancu. On može biti ≤ 1 ili > 1.

Najgori slučaj u ulančanoj metodi bi bio: n ključeva koji se vežu na istu poziciju, tvoreći listu duljine n. Vrijeme za pretraživanje u najgorem slučaju je $\Theta(n)$ + vrijeme potrebno za obrađivanje hash funkcije, što nije ništa bolje od korištenja povezane liste za sve elemente.

Prosječna izvedba ove metode ovisi o tome koliko dobro hash funkcija h raspoređuje skup ključeva po m pozicija. Pretpostavimo da svi elemeneti, neovisno jedan o drugom, imaju jednaku vjerojatnost biti pridruženi bilo kojoj od m pozicija, tako zvana pretpostavka jednostavnog uniformnog hashiranja.

Označimo duljinu liste $T[j]$ s n_j. Za $j = 0, 1, \ldots, m-1$ takav da je $n = n_0 + n_1 + \cdots + n_{m-1}$, i srednja vrijednost od n_j je $E[nj] = \alpha = n/m$.

Pretpostavljamo da se hash vrijednost $h(k)$ može izračunati u vremenu $O(1)$, tako da vrijeme potrebno za pretraživanje elementa s ključem k ovisi linearno o duljini $n_{h(k)}$ liste $T[h(k)]$. Zanemarimo vrijeme, $O(1)$, potrebno za izvršavanje hash funkcije i za pristup poziciji $h(k)$, razmotrimo očekivani broj elemenata u listi $T[h(k)]$ čiji se ključ uspoređuje s k, kako bi se vidjelo jesu li jednaki. Trebamo razmotriti dva slučaja. U prvom, pretraga je neuspješna, tj. niti jedan element tablice ne sadrži ključ k,a u idućem, pretragom pronalazimo element s ključem k.

Teorem 3.1. *U hash tablici, u kojoj se sudari razrješavaju ulančavanjem, neuspješna i uspješna pretraga traju $\Theta(1 + \alpha)$, uz pretpostavku jednostavnog uniformnog hashiranja*

Zaključujemo da, ukoliko je broj pozicija hash tablice, u najmanju ruku, proporcionalan broju elemenata tablice, imamo $n = O(m)$, a potom i $\alpha = n/m = O(m)/m = O(1)$, stoga pretraga, u prosjeku, zahtjeva konstantno vrijeme. Pošto je za unos potrebno $O(1)$, u najgorem slučaju, kada su liste dvostruko povezane, sve operacije rječnika se mogu, u prosjeku, izvršiti u $O(1)$ vremenu.

4 Hash funkcije

Dobra hash funkcija zadovoljava pretpostavku da svaki ključ ima jednaku vjerojatnost da završi na bilo kojoj poziciji (od njih m), neovisno o drugim ključevima. Uglavnom ovaj uvjet nije moguće provjeriti jer ćemo rijetko znati vjerojatnosnu distribuciju prema kojoj se ključevi izvlače. Ponekad će nam distribucija biti poznata. Za primjer možemo proučiti slučaj kada znamo da su ključevi nasumični realni brojevi, k, koji se neovisno i ravnomjerno raspoređuju u rasponu $0 \leq k < 1$, pri čemu hash funkcija izgleda kao $h(k) = \lfloor km \rfloor$ i ona zadovoljava gore spomenut uvjet jednostavnog uniformnog hashiranja.

4.1 Interpretacija ključeva kao prirodnih brojeva

Kod većine hash funkcija se pretpostavlja da je skup ključeva skup prirodnih brojeva s nulom $\mathbb{N}_0 = \{0, 1, 2, \dots\}$, stoga ako ključevi nisu prirodni brojevi, potrebno je pronaći način da se oni interpretiraju preko prirodnih brojeva. Na primjer, string načinjen od slova se može interpretirati preko prirodnih brojeva pomoću zapisa u bazi 128. Recimo, 'pt' se može interpretirati kao par (112, 116) jer je p = 112, a t = 116 u ASCII zapisu. Pošto su brojevi 112 i 116 iz baze 128, imamo $(112 \cdot 128) + 116 = 14452$.

U nastavku ćemo pretpostaviti da su ključevi prirodni brojevi.

4.2 Metoda dijeljenja

U metodi dijeljenja, za stvaranje hash funkcija pridružujemo ključ k jednom od m polja, uzimanjem ostatka pri dijeljenju broja m s k, tj. hash funkcija ima oblik $h(k) = k \bmod m$.

Na primjer, ukoliko je hash tablica veličine $m = 12$, a ključ $k = 100$, tada $h(k) = h(100) = 100 \bmod 12 = 4$ ili, ako uzmemo $m = 20$ i $k = 91$ dobit ćemo $h(k) = h(91) = 91 \bmod 20 = 11$. Pošto funkcija zahtjeva samo jednu operaciju dijeljenja, metoda podjele je izuzetno brza.

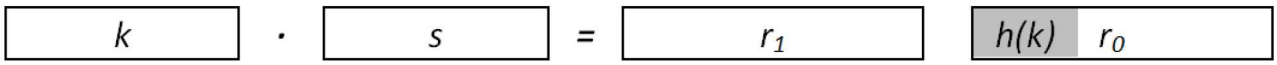
Pri korištenju metode dijeljenja, uobičajeno nastojimo izbjeći određene vrijednosti varijable m. Recimo, m ne bi trebao biti potencija broja 2 jer kada je $m = 2^p$ tada $h(k)$ samo vraća zadnjih p bitova od k. Uzmimo $m = 2$ i $k = 10110_2$, tada je $h(k) = k \bmod m = 0$ ili $m = 2^6$ i $k = 1011000111011010_2$, u ovom slučaju $h(k)$ vraća 011010. Bolje je napraviti hash funkciju ovisnu o svim bitovima od k.

Prost broj koji nije preblizu potencije broja 2 je često dobar izbor za m. Za primjer, pretpostavimo da želimo raspodijeliti hash tablicu, u kojoj se sudar razriješio lančanjem, tako da sadrži stringove od približno n = 2000 znakova, pri čemu znak ima 8 bitova. S isptivanjem prosječno triju elemenata u neuspjelom pretraživanju, napravimo tablicu veličine m = 701. Broj 701 smo dobili odabirom prostog broja u blizini broja 2000/3, ali koji se ne nalazi u blizini potencije broja 2. Tretiranjem svakog ključa k kao cijelog broja, naša hash funkcija bi izgleda kao $h(k) = k \bmod 701$.

4.3 Multiplikativna metoda

Korištenjem multiplikativne metode, hash funkcije se stvaraju kroz dva koraka. Prvo, pomnožimo ključ k s konstantom A, koja je u rasponu $0 < A < 1$, i izvučemo djelomični dio od kA. Potom, pomnožimo dobivenu vrijednost s m i uzmemo donju među od dobivenog. Iz toga slijedi da hash funkcija ima oblik $h(k) = \lfloor m(kA \bmod 1) \rfloor$, pri čemu "$kA$ mod 1" znači djelomični dio od kA, odnosno kA - $\lfloor kA \rfloor$.

Uobičajeno m biramo kao potenciju broja 2. Pretpostavimo da je veličina riječi u memoriji w bitova i da k stane u jednu riječ. Ograničimo A da bude razlomak oblika $s/2^w$, gdje je s u rasponu $0 < s < 2^w$. Prvo pomnožimo k s w-bitnim cijelim brojem $s = A \cdot 2^w$. Rezultat je $2w$-bitna vrijednost oblika $r_1 \cdot 2^w + r_0$, gdje je r_1 riječ visokog, a r_2 riječ nižeg reda produkta. Željena p-bitna hash vrijednost je načinjena od p najznačajnijih bitova od r_0. Vidi sliku 4.1.



Slika 4.1: Multiplikativna metoda

Primjer 4.1. $n = 8$, $p = 3$, $w = 5$, $k = 21$

Odaberimo s tako da vrijedi $0 < s < 2^5$, *npr.* $s = 13$, *što će nam dati:*
$A = s/2^w = 13/2^5 = 13/32$
Po formuli dobivamo:
$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor 8(21 \cdot 13/32 \bmod 1) \rfloor = 4.$
Do istog rezultata možemo doći i na sljedeći način. Pogledajmo $k \cdot s = r_1 \cdot 2^w + r_0$.
Kada uvrstimo $k = 21$, $s = 13$ *i* $w = 5$, *dobivamo:*
$273 = 8 \cdot 2^5 + 17$, *što znači, da su* $r_1 = 8$ *i* $r_0 = 17$.
$r_0 = 17 = 10001_2$, *pošto je* $p = 3$, *uzimamo prve 3 signifikantne znamenke pa imamo* $h(k) = 100_2 = 4$.

Iako ova metoda radi sa svakom vrijednošću konstante A, ne radi sa svima jednako dobro. Optimalan izbor ovisi o karakteristikama tipa podatka koji koristimo.

5 Otvoreno adresiranje

Kod otvorenog adresiranja svi elementi su spremljeni u hash tablicu, odnosno svaki tablični ulaz sadrži ili element dinamičkog skupa ili NIL. Prilikom traženja elementa, sistematično ispitujemo pozicije, dok se željeni element ne pronađe ili dok se ne ustanovi da tog elementa nema u tablici. Radimo bez lista i bez elemenata spremljenih van tablice, kao kod ulančavanja. Prema tome, u otvorenom adresiranju hash tablica se može "popuniti" tako da se onemogući danji unos. Jedna od posljedica je ta da faktor opterećenja α ne može prekoračiti 1.

Ukoliko želimo, možemo pohraniti povezanu listu za povezivanje unutar hash tablice, ali bi to bilo u kontradikciji sa samom idejom metode otvorenog adresiranja, koja nastoji zaobići uopće korištenje pokazivača. Umjesto praćenja pokazivača, računamo slijed polja koje će biti potrebno ispitati. Višak memorije oslobođen zbog nekorištenja, odnosno nespremanja, pokazivača dopušta korištenje hash tablice s većim brojem polja za istu količinu memorije.

Prilikom unosa, pomoću metode otvorenog adresiranja, mi uspješno ispitujemo hash tablicu dok ne pronađemo prazno poziciju u koju stavljamo ključ. Umjesto održavanja fiksnog reda $0, 1, \ldots, m-1$, čije pretraživanje zahtjeva $\Theta(n)$ vremena, niz pozicija koje pretražimo ovisi o ključu koji unosimo. Pri određivanju koje pozicije je potrebno preispitati, potrebno je proširiti hash funkciju tako da sadrži "ispitivački" broj (koji kreće od 0), kao drugi unos. S ovom promjenom, naša hash funkcija poprima oblik $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$.

Prilikom korištenja otvorenog adresiranja, zahtjeva se da za svaki ključ k, "ispitivački" niz $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ treba biti permutacija od $\langle 0, 1, ..., m-1\rangle$, tako da se svako polje hash tablice, kako se ona popunjava, smatra kao pozicija za novi ključ. U idućem algoritmu pretpostavljamo da su elementi hash tablice T ključevi bez dodatne informacije,tj. ključ k je identičan elementu koji sadrži ključ k. Svaka pozicija sadrži ili ključ ili NIL, u slučaju da je pozicija prazna. Funkcija HASH-INSERT (algoritam 7) prima dvije vrijednosti, hash tablicu T i ključ k. Ona će vratiti, ili broj pozicije na koju je pohranila proslijeđeni ključ k, ili vraća pogrešku jer je dana hash tablica popunjena.

Algoritam 7

```
function HASH-INSERT(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] == NIL then
            T[j] = k
            return j
        else
            i = i + 1
    until i == m
    error "Hash tablica je puna!"
```

Algoritam za pretraživanje ključa k ispituje isti niz polja koje je algoritam unosa prošao prilikom unosa ključa k, stoga pretraga može završiti neuspješno kada naiđe na prazno polje, pošto bi k bio unesen na to mjesto, umjesto kasnije u nizu. Bitno je napomenuti da se ovdje smatra kako ključevi nisu brisani iz hash tablice. Funkcija HASH-SEARCH (algoritam 8) prima dvije varijable, hash tablicu T i ključ k, potom vraća vrijednost j, ukoliko pronađe polje j koje sadrži traženi ključ k ili, u suprotnom, vraća NIL.

Algoritam 8

```
function HASH-SEARCH(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] == k then
            return j
        i = i + 1
    until T[j] == NIL or i == m
    return NIL
```

Brisanje iz hash tablice postaje teško kada se koristimo metodom otvorenog adresiranja. Kada obrišemo ključ iz polja i, ne možemo jednostavno označiti polje kao prazno, spremajući u njega NIL. Ukoliko to učinimo, moguće je da nećemo moći povratiti bilo koji ključ k tokom čijeg unosa smo morali ispitati polje i i zaključili da je zauzeto. Ovaj problem možemo riješiti označavajući polja, preciznije pohranjivanjem posebne vrijednosti DELETED unutar njih, umjesto NIL. Tada bismo nadogradili funkciju HASH-INSERT tako da tretira takvo polje kao prazno, kako bismo mogli unijeti novi ključ. Nemamo potrebu za promjenom funkcije HASH-SEARCH jer će ona preći preko nove vrijednosti, DELETED, prilikom pretraživanja. Bitno je napomenuti da uporabom nove vrijednosti DELETED vrijeme pretraživanja više ne ovisi o faktoru opterećenja α i iz tog razloga je ulančavanje učestalije korištena metoda razrješavanja sudara kada se ključevi moraju brisati.

U našoj analizi, koristiti ćemo pretpostavku uniformnog hashiranja: "ispitivački" niz svakog ključa je jednako vjerojatan da bude bilo koja od m! permutacija od $\langle 0, 1, \ldots, m-1 \rangle$.

Razmotrit ćemo tri često korištene metode pri određivanju "ispitivačkog" niza, potrebnog prilikom otvorenog adresiranja:linearno ispitivanje, kvadratno ispitivanje i dvostruko hashiranje. Te metode garantiraju da je $\langle h(k,0), h(k,1), ..., h(k,m-1) \rangle$ permutacija od $\langle 0, 1, ..., m-1 \rangle$ za svaki ključ k. Niti jedna od ovih metoda ispunjava uvjet uniformnog hashiranja, pošto niti jedna od njih nije u stanju generirati više od m^2 drugačijih "ispitivačkih" nizova.

Dvostruko hashiranje ima najveći broj "ispitivačkih" nizova i, uz to, daje najbolje rezultate.

5.1 Linearno ispitivanje

Uz običnu hash funkciju $h' : U \to \{0, 1, \ldots, m-1\}$, koju nazivamo pomoćna hash funkcija, metoda linearnog ispitivanja koristiti hash funkciju $h(k, i) = (h'(k) + i)$ mod m za $i = 0, 1, \ldots, m-1$. Uz dani ključ k, prvo ispitamo $T[h'(k)]$, tj. polje dobiveno pomoćnom hash funkcijom, potom ispitamo $T[h'(k) + 1]$ pa sve do polja $T[m-1]$. Potom $T[0], T[1], \ldots$ sve dok konačno ne ispitamo polje $T[h'(k) - 1]$. Postoji samo m različitih "ispitivačkih" sljedova jer početni određuje cijeli slijed.

Linearno ispitivanje se jednostavno implementira, ali se javlja problem koji nazivamo primarno grupiranje. Dugi nizovi zauzetih pozicija se s vremenom nagomilaju, što uzrokuje povećano prosječno vrijeme pretraživanja. Grupacija se javlja jer će se prazna pozicija, kojoj prethodi i popunjenih pozicija, iduća popuniti s vjerojatnošću $(i+1)/m$.

Primjer 5.1. *Proučimo metodu na primjeru $h(k) = k$ mod 7 i ključevima 50, 700, 76, 85, 92.*

a) Započinjemo s praznom tablicom veličine 7

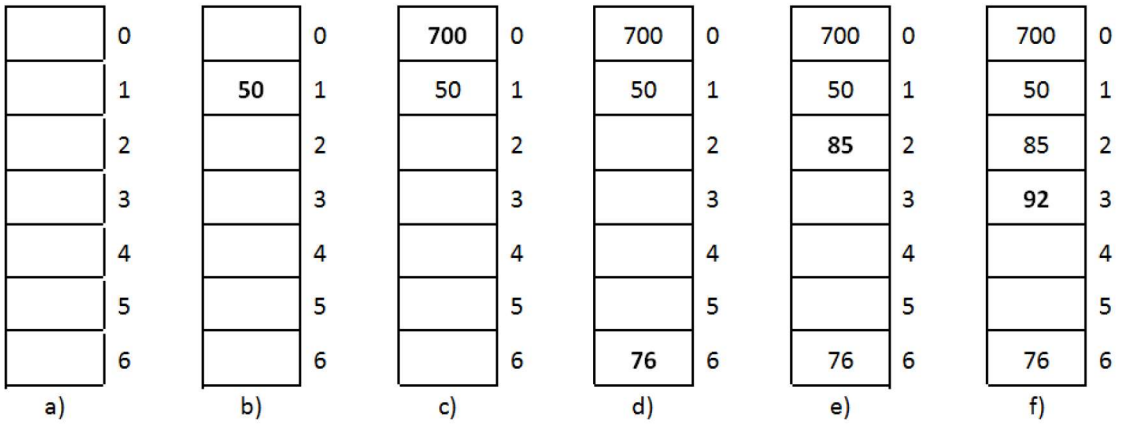
b) $h(50) = 1$

c) $h(700) = 0$

d) $h(76) = 6$

e) $h(85) = 1$, *ali pošto se na poziciji 1 nalazi neki element, 85 ćemo spremiti na iduću slobodnu poziciju, a to je 2*

f) $h(92) = 1$, *pozicija 1 je zauzeta pa se pomičemo na iduću slobodnu, a to je 3*



5.2 Kvadratno ispitivanje

Kvadratno ispitivanje koristi hash funkciju oblika $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, gdje je h pomoćna hash funkcija, c_1 i c_2 su pozitivne pomoćne konstante i $i = 0, 1, \ldots, m-1$. Početna ispitana pozicija je $T[h'(k)]$, dok su danje ispitane pozicije pomaknute s kvadratnom ovisnošću prema broju i. Ova metoda funkcionira bolje od metode linearnog ispitivanja, ali kako bismo u potpunosti iskoristili hash tablice, vrijednosti c_1, c_2 i m su ograničene. Isto tako, ukoliko dva ključa imaju istu početnu "ispitanu" poziciju tada su njihovi "ispitivački" nizovi isti jer $h(k_1, 0) = h(k_2, 0)$ povlači $h(k_1, i) = h(k_2, i)$. Ovo svojstvo nas dovodi do blagog oblika grupiranja, to nazivamo sekundarno grupiranje. Kao i kod linearnog ispitivanja, početno "ispitano" polje određuje čitav slijed, stoga se koristi samo m različitih "ispitivačkih" sljedova.

Primjer 5.2. *Proučimo metodu na primjeru $h(k) = k \bmod 7$ i ključevima 76, 40, 48, 5, 55.*

a) Započinjemo s praznom tablicom veličine 7

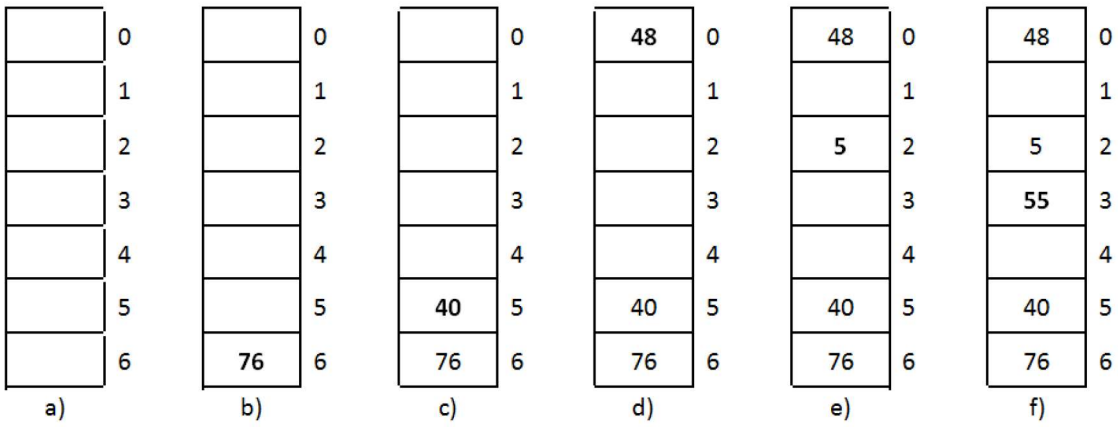
b) $h(76) = 6$

c) $h(40) = 5$

d) $h(48) = 6$, ali pošto se na poziciji 6 nalazi neki element, gledamo $(48 + 1^2) \bmod 7$ i dobivamo 0

e) $h(5) = 5$, ali pošto se na poziciji 5 nalazi 40, gledamo $(5 + 1^2) \bmod 7$ i dobivamo 6, tamo se nalazi 76 pa računamo $(5 + 2^2) \bmod 7 = 2$

f) $h(55) = 6$, $(55 + 1^2) \bmod 7 = 0$, $(5 + 2^2) \bmod 7 = 3$



5.3 Dvostruko hashiranje

Dvostruko hashiranje nudi jednu od najboljih metoda za otvoreno adresiranje jer dobivene permutacije imaju mnoge karakteristike nasumično odabranih permutacija. U ovoj metodi koristimo hash funkciju oblika $h(k,i) = (h_1(k) + ih_2(k))$ mod m, gdje su i h_1 i h_2 pomoćne hash funkcije. Početna "ispitivačka" pozicija postaje $T[h1(k)]$, dok su sljedeće odstupanja od prethodne za $h_2(k)$ modulo m. Za razliku od linearnog ili kvadratnoga ispitivanja, "ispitivački" slijed, u ovoj metodi, ovisi o ključu k na dva načina.

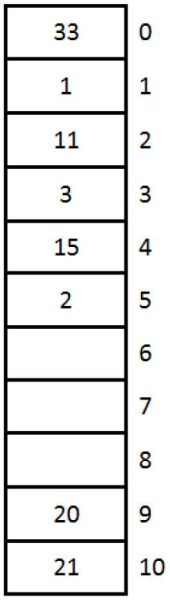
Vrijednost $h_2(k)$ mora biti relativno prosta u odnosu na veličinu hash tablice, u oznaci m, kako bi se cijela tablica mogla pretražiti. Pogodan način osiguravanja ovog uvjeta je da m postavimo kao potenciju broja 2, a h_2 oblikujemo tako da uvijek bude neparan broj. Drugi način je da m bude prost, a h_2 takav da uvijek bude pozitivan cijeli broj manji od m. Za primjer možemo uzeti da je m prost i
$h_1(k) = k$ mod m
$h_2(k) = 1 + (k$mod $m')$,
gdje je m' odabran kao broj malo manji od m, recimo $m - 1$. Na primjer, ako je k = 123456, m = 701 i m' = 700, dobivamo $h_1(k) = 80$ i $h_2(k) = 257$. Tako da prvo ispitamo poziciju 80, a potom provjerimo svako 257. polje (modulo m) dok ne pronađemo ključ ili dok ne provjerimo svako polje.

Kada je m prost ili potencija broja 2, dvostruko hashiranje se unapređuje, u usporedbi s linearnim i kvadratnim isprobavanje, u tome što se koriste $\Theta(m^2)$ ispitanih sljedova, umjesto $\Theta(m)$, jer svaki mogući par $(h_1(k), h_2(k))$ stvara drugačiji ispitni slijed. Kao rezultat, za takve vrijednosti varijable m, izvedba dvostrukog hashiranja se doima izuzetno blizu izvedbi "idealne" sheme uniformnog hashiranja.

Iako varijabla m može poprimiti i druge vrijednosti, izuzev prostih brojeva i potencija broja 2, prilikom korištenja ove metode, ali tada u praksi postaje znatno teže učinkovito generirati $h_2(k)$ tako da bude relativno prost u odnosu na m.

Primjer 5.3. *Proučimo primjer s pomoćnim funkcijama* $h_1(k) = k$ *mod* m *i* $h_2(k) = 1 + (k$ *mod* $(m-1))$, $m = 11$ *i ključevima* 15, 3, 20, 21, 33, 11, 1, 2.

$h_1(15) = 4$
$h_1(3) = 3$
$h_1(20) = 9$
$h_1(21) = 10$
$h_1(33) = 0$
$h_1(11) = 0$, *pošto je 0 zauzeto gledamo* $h_1(11) + h_2(11) = 0 + 2 = 2$
$h_1(1) = 1$
$h_1(2) = 2$, *pošto je 2 zauzeto gledamo* $h_1(2) + h_2(2) = 2 + 3 = 5$
Slika 5.3 prikazuje rezultat primjera.



Slika 5.3: Rezultat primjera 5.3

5.4 Analiza otvorenog adresiranja

Kao i kod analize ulančavanja, u analizi otvorenog adresiranja fokusiramo se na faktor opterećenja, α = n/m, hash tablice. Naravno, uz otvoreno adresiranje, najviše jedan element zauzima pojedinu poziciju, stoga $n \leq m$, što povlači $\alpha \leq 1$.

Pretpostavljamo da rabimo uniformno hashiranje. U ovoj idealnoj shemi, ispitani slijed $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ koji se koristi pri unosu ili pretrazi bilo kojeg ključa k je jednako vjerojatan da bude bilo koja permutacija od $\langle 0,1,\ldots,m-1\rangle$. Danom ključu pripada jedinstveni ispitni slijed.

Sada ćemo analizirati očekivan broj "ispitnih" pozicija uz korištenje otvorenog adresiranja uz pretpostavku uniformnog hashiranja, započinjući analizom broja ispitanih pozicija u neuspjelom pretraživanju.

Teorem 5.1. *Neka imamo hash tablicu uz otvoreno adresiranje s faktorom opterećenja $\alpha = n/m < 1$, očekivan broj ispitanih pozicija u neuspjelom pretraživanju je maksimalno 1/(1 - α), uz pretpostavku uniformnog hashiranja.*

Dokaz. U neuspješnom pretraživanju, svaka ispitana pozicija, izuzev posljednje, pristupa zauzetom polju koje ne sadrži željeni ključ i posljednje ispitano polje je prazno. Definirajmo nasumičnu varijablu X, kao broj "ispitivanja" obavljenih u neuspjelom pretraživanju. Potom, definirajmo događaj A_i, za $i = 1,2,\ldots,$ kao događaj "ispitivanja" zauzetog polja. Tada događaj {X $\geq$ i} je presjek događaja $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. Povezat ćemo P(X $\geq$ i), povezivanjem P($A_1 \cap A_2 \cap \cdots \cap A_{i-1}$).

$$P(A_1 \cap A_2 \cap \cdots \cap A_{i-1}) = P(A_1) \cdot P(A_2|A_1) \cdot P(A_3|A_1 \cap A_2) \ldots P(A_{i-1}|A_1 \cap A_2 \cap \cdots \cap A_{i-2}). \quad (1)$$

Pošto imamo n elemenata i m polja, $P(A_1) = n/m$. Za $j > 1$, vjerojatnost da postoji j. ispitana pozicija i da je riječ o zauzetom polju, uz to da su prvih $j-1$ ispitanih pozicija bilo zauzeto, je $(n-j+1)/(m-j+1)$. To smo dobili jer bismo pronalazili jednog od preostalih $(n-(j-1))$ elemenata u jednom od $(m-(j-1))$ neispitanih polja, s pretpostavkom uniformnog hashiranja, vjerojatnost je omjer tih vrijednosti. Pošto $n < m$ implicira $(n-j)/(m-j) \leq n/m$, za svaki j takav da $0 \leq j < m$, imamo da za svaki i takav da $1 \leq i < m$,

$$P(X \geq i) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \cdots \cdot \frac{n-i+2}{m-i+2} \leq (\frac{n}{m})^{i-1} = \alpha^{i-1}. \tag{2}$$

Potom,

$$\begin{aligned} E[X] &= \sum_{i=1}^{\infty} P(X >= i) \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=1}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha}. \end{aligned}$$

□

Ova povezanost, $\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$, ima intuitivnu interpretaciju.

Uvijek postoji prvo ispitano polje. S vjerojatnošću približno α, prvo ispitano polje biva zauzeto, stoga moramo ispitati još jedno. S vjerojatnošću približno α^2, prva dva polja su zauzeta tada trebamo ispitati treće, itd.

Ukoliko je α konstanta, Teorem 4.1 predviđa da će vrijeme neuspješnog pretraživanja biti $O(1)$. Na primjer, ako je pola hash tablice popunjeno, prosječni broj ispitanih pozicija je maksimalno 1/(1 - 0.5) = 2. Ukoliko je 90 posto tablice popunjeno, prosječni broj ispitanih pozicija je maksimalno 1/(1 - 0.9) = 10.

Korolar 5.2. *Unos elementa u otvoreno adresiranu hash tablicu s faktorom opterećenja α zahtjeva najviše 1/(1 - α) ispitanih pozicija, uz pretpostavku uniformnog hashiranja.*

Dokaz. Element je unesen jedino, ako postoji slobodan prostor u tablici, stoga $\alpha < 1$. Unos ključa zahtjeva neuspjelo pretraživanje popraćeno unosom ključa u prvo slobodno pronađeno mjesto. Tako je očekivani broj ispitanih pozicija maksimalno 1/(1 - α).

□

Teorem 5.3. *Neka je dana otvoreno adresirana tablica s faktorom opterećenja $\alpha < 1$,tada je očekivan broj ispitanih pozicija u uspješnom pretraživanju maksimalno*

$$\frac{1}{\alpha} ln \frac{1}{1-\alpha}, \tag{3}$$

uz pretpostavke uniformnog hashiranja i pretpostavku da svaki ključ tablice ima jednaku vjerojatnost biti pretražen.

Dokaz. Pretraživanje ključa k stvara isti slijed ispitanih pozicija kao i kada unosimo element s ključem k. Ako je k bio $(i+1)$. uneseni ključ u hash tablicu, očekivan broj ispitanih pozicija tokom pretrage za ključ k je najviše $1/(1-i/m) = m/(m-i)$. U prosjeku, gledajući svih n ključeva hash tablice, dobivamo:

$$\begin{aligned}
\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\
&= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} \frac{1}{k} \\
&\leq \frac{1}{\alpha} \int_{m-n}^{m} 1/x dx \\
&= \frac{1}{\alpha} ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} ln \frac{1}{1-\alpha}.
\end{aligned}$$

□

Ukoliko je hash tablica puna, očekivan broj ispitanih pozicija u uspješnom pretraživanju je manji od 1.387. Ukoliko je 90 posto hash tablice popunjeno, očekivan broj ispitanih pozicija je manji od 2.559.

6 Implementacija hash tablice

Proučimo jednostavan kod u programskom jeziku python, kojim se demonstrira stvaranje i popunjavanje hash tablice. Program će, ukoliko dva ili više elementa budu pridruženi istoj poziciji naše tablice, elemente spremati u slijed, tako da svaki element ima svoga sljedbenika.

```
class elem: #element hash tablice
#prima kljuc i vrijednost

    def __init__(self, k, v):
        self.k = k #kljuc
        self.value = v #vrijednost
        self.next = None #sljedbenik



class HashTable: #klasa hash tablice
#m je varijabla za velicinu tablice

    def __init__(self, m):
        self.m = m
        #stvori praznu listu duljine m
        self.lista = [None] * self.m


    def unos(self, k, value):
    #funkcija unosa, prima kljuc k i neku vrijednos

        #uzimamo hash vrijednost kljuca
        kHashed = hash(k)
        #pozicija na koju spremano elem
        #ukoliko je hHashed > m, % ga smanjuje
        p = kHashed % self.m

        #slucaj kada je ta pozicija slobodna
        if self.lista[p] is None:
            self.lista[p] = elem(k, value)

        #slucaj kada je ta pozija zauzeta
        else:
            self.podlista(k, value, self.lista, p)
```

```
def podlista(self, k, value, lista, p):
#pravi listu na poziciji u hash tablici

    #zeljeni cvor
    node = lista[p]

    while node is not None:
        #ukoliko su kljucevi isti
        if node.k == k:
            node.value = value
            node = None

        #ukoliko na iducoj poziciji nema nista
        elif node.next is None:
            #sprema elem na iducu poziciju
            node.next = elem(k, value)
            node = None

        #pomicemo se na iducu poziciju, pa gledamo
        #podudaraju li se kljucevi ili ne postoji njen sljedbenik
        else:
            node = node.next


def trazi(self, k):
#trazi poziciju s kljucem k te vraca vrijednost koja se tamo nalazi

    #hash vrijednost kljuca
    kHashed = hash(k)
    #pozicija na kojoj bi se kljuc trebao nalaziti
    p = kHashed % self.m

    #ukoliko na toj poziciji nesto postoji
    if self.lista[p] is not None:
        #postavljanje cvora
        node = self.lista[p]

        #cvorom prolazimo po podlisti
        while node is not None:

            #uspjesni pronalazak
            if node.k == k:
                return node.value
            #ukoliko na trenutnoj poziciji nije elem.
            #s trazenim kljucem, pomicemo se na iduci u podlisti
            node = node.next
```

```
if __name__ == "__main__":

    #stvaramo hash tablicu s 10 pozicija
    T = HashTable(10)

    #na prviu poziciju stavljamo elem s vrijednoscu 55
    T.unos(0,55)

    #trazimo vracanje vrijednosti koja je pod kljucem 0
    print(T.trazi(0))

    #na trecu poziciju stavljamo elem s vrijednoscu 1000
    T.unos(2,1000)

    #na trecu(jer je 22%10=2) poziciju stavljamo
    #elem s vrijednoscu 55577
    T.unos(22,55577)

    #trazimo vracanje vrijednosti koja je pod kljucem 22
    print(T.trazi(22))

    #trazimo vracanje vrijednosti koja je pod kljucem 2
    print(T.trazi(2))

    #na trecu(jer je 98765432%10=2) poziciju stavljamo
    #elem s vrijednoscu 13
    T.unos(98765432,13)

    #sada cemo ispisati vrijednosti svih elemenata
    #koji se nalaze na poziciji s indeksom 2 u tablici

    #cvor je trenuto postavljen na prvi element
    #koji se nalazi na trecoj poziciji u tablici
    node = T.lista[2]

    while node is not None:
        print(node.value)
        node = node.next
```

Rezultat ovog koda izgleda ovako:

```
55
55577
1000
1000
55577
13
```

Literatura

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algoritms, 2Ed, MIT Press, 2001

[2] R. Sedgewick, Algorithms in C, Parts 1-4 Fundamentals, Data Structure, Sorting, Searching, Third Edition, 1997.

[3] M. T. Goodrich, R. Tamassia, D. M. Mount, Data Structures and Algorithms in C++, Wiley, 2010.

[4] A. Drozdek, Data Structures and Algorithms in C++, Cengage Learning, 2012.

[5] R. Sedgewick, K. Wayne, Algorithms, Addison-Wesley Professional, 2011.

[6] M. J. Atallah, Algorithms and Theory of Computation Handbook, CRC Press, 1998.