



STRUKTURE PODATAKA

LETNJI SEMESTAR

HASH TABLICE

Prof. Dr Leonid Stoimenov
Katedra za računarstvo
Elektronski fakultet u Nišu

PREGLED PREDAVANJA

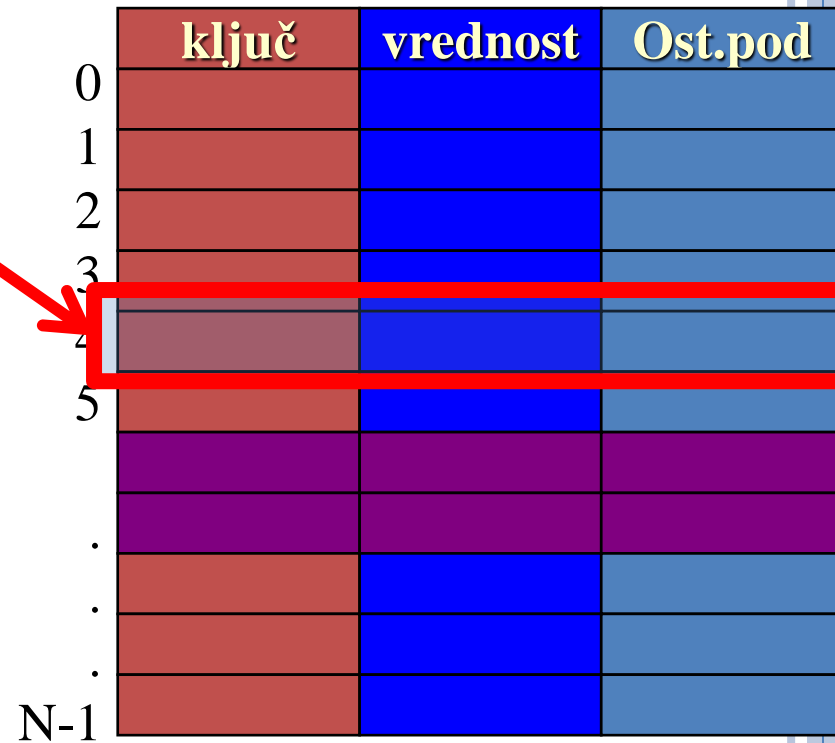
- Uvod – šta su Hash tablice
- Osnovni pojmovi
- Hash funkcija
- Kolizija i način rešavanja



UVOD - ŠTA JE HASH TABLICA

- **Hash tablica** je osnovna struktura podataka za smeštanje i pretragu podataka
- **Hash tablica** je struktura podataka koja omogućava
 - **direktan pristup** podacima ...
 - ... izračunavanjem njihovog položaja u tablici ...
 - ... na osnovu vrednosti ključa.

k



	ključ	vrednost	Ost.pod
0			
1			
2			
3			
4			
5			
.			
.			
.			
N-1			

UVOD - ŠTA JE HASH TABLICA

Hash tablica

- generalizacija običnog polja

indeksira se
celobrojnim
indeksima

- kao **polje indeksirano proizvoljnim ključevima**

mogu biti:

→ brojevi,

→ nizovi znakova, i sl.

	ključ	vrednost	Ost.pod
0			
1			
2			
3			
4			
5			
.			
.			
.			
N-1			

UVOD - ŠTA JE HASH TABLICA

○ Cilj:

- smeštanje/dodavanje i
- brzo pretraživanje **velike količine podataka**.

○ Osnovna ideja:

- da se ulazni skup veličine N (koji može biti jako veliki; veći od raspoložive memorije)
- svede na manji broj, n , stavki.

○ Takvu “kompresiju” radi **hash funkcija**

	ključ	vrednost	Ost.pod
0			
1			
2			
3			
4			
5			
N-1			

ELEMENT TABELE – KLJUČ I PODATAK

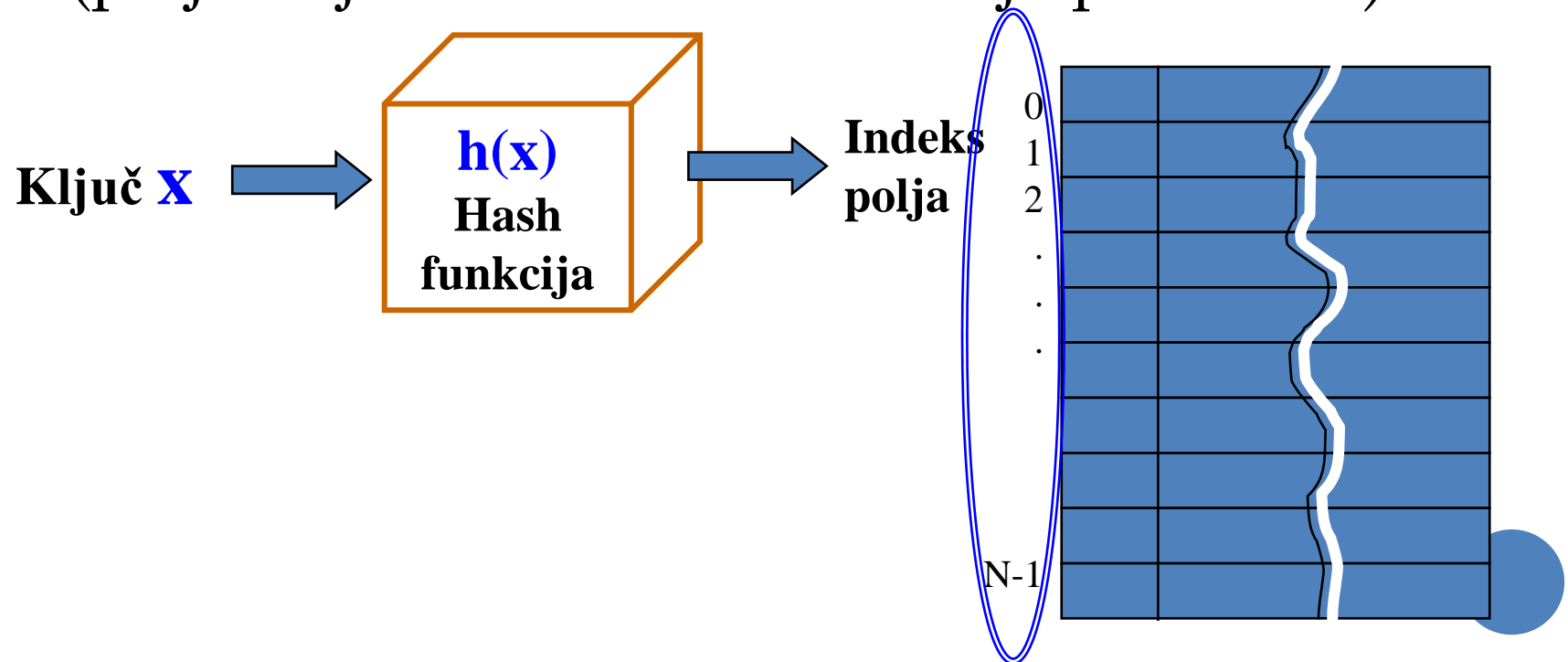
- Zbog traženja, u svakoj **ćeliji tabele** odnosno u svakom **elementu tabele**, čuva se
 - ključ i odgovarajući podatak,
 - ... ali **odvojeno**, osim kada je vrednost ključa deo podatka.

Ćelija / Element Tabele	ključ	podatak
	“Petar”	“Perić”, “ul. Perina 123”, “9675846 6554 545”
	“Joca”	“Jocić”, “ul. Jabuka 321”, “0044 1970 622455”




ILUSTRACIJA RADA HASH FUNKCIJE

- Hash funkcija $h(x)$ vrši preslikavanje vrednosti ključa x u **adrese** (indekse) tablice (polja koje se koristi za čuvanje podataka)



HASH FUNKCIJA

- **Ključ** hash tablice - vrednost podatka na osnovu koga se vrši smeštanje u hash tablicu.
 - **Hash funkcija** - Funkcija koja preslikava vrednosti **ključeva** (vrednosti na osnovu kojih se smeštaju podaci) u **cele brojeve**, tj. vrednosti indeksa polja.
 - Hash funkcija omogućava mehanizam pristupa koji **izbegava pretraživanje** strukture radi nalaženja elementa.
 - Na osnovu **vrednosti podataka** vrši se **izbor funkcije** za preslikavanje njihove vrednosti u adrese, odnosno indekse polja.
 - Njen izbor zahteva izvesno predznanje o vrednostima podataka koje se smeštaju u *hash* tablicu
- 

FORMALNA DEFINICIJA HASH FUNKCIJE

- Hash funkcija h je preslikavanje iz skupa vrednosti ključa U ($|U| = N$, N veličina hash tablice) u skup indeksa hash tablice:

$$h : U \rightarrow \{0, 1, \dots, N-1\}$$

- Za funkcionisanje ove ideje važno je da hash funkcija $h(\text{el_type})$ **ravnomerno raspoređuje** vrednosti iz skupa el_type na indekse tablice $0, 1, 2, \dots, N-1$.
- To ne može da bude uvek slučaj!!
- Svrha** hash funkcije je da preslika ključeve iz nekog opsega u vrednosti indeksa ali tako da se vrednosti ključeva distribuiraju **slučajno** u celom opsegu indeksa tablice.
- Ključevi mogu biti kompletno slučajni ili delimično slučajni.

ZAHTEVI KOD IZBORA HASH FUNKCIJE

○ Perfektna *hash* funkcija??

- **Cilj**: Izabrati dobru, odnosno što je moguće bolju *hash* funkciju
- **Osnovni zahtevi** koje treba da ispuni dobra *hash* funkcija su:
 - da izbegava kolizije,
 - da rasipa vrednosti ravnomerno po čitavoj tablici i
 - da se jednostavno (tj. brzo) izračunava.



ZAHTEVI KOD IZBORA HASH FUNKCIJE

- Od hash funkcije jako **zavisi brzina** umetanja i traženja elemenata u hash tablici.
- Dobra hash funkcija treba da izvrši preslikavanje ključeva u indekse što **slučajnije** i **uniformnije** jer u tom slučaju ima najmanje kolizija.
- U idealnom slučaju su ubacivanje i traženje elementa složenosti **$O(1)$** .
- Ovo važi **posle računanja Hash funkcije** - složenost njenog računanja može da zavisi od dužine ključa.



METODE ZA IZRAČUNAVANJE HASH FUNKCIJE

- Najčešće metode za izračunavanje *hash* funkcija su sledeće:

- metod **deljenja**,

$$h(k) = k \bmod M$$

- metod **sredine kvadrata**,

$$h(k) = \left\lfloor \frac{M}{W} (k^2 \bmod W) \right\rfloor$$

- metod **množenja**,

- **Fibonačijev** metod,

$$h(k) = \left\lfloor \frac{M}{W} (ak \bmod W) \right\rfloor$$

- metod **presavijanja**,

- metod **ekstrakcije** (izvlačenja) i

- metod **transformacije osnove**.

**Više informacija:
PRAKTIKUM!!**



PRIMER RADA HASH FUNKCIJE

- Hash funkcija

$$h = (\text{key}) \bmod 10$$

Ključ **h**

- 81 1

- 64 4

- 36 6

- 49 9

- Nema kolizije !!

- 61 ?

- 34 ?

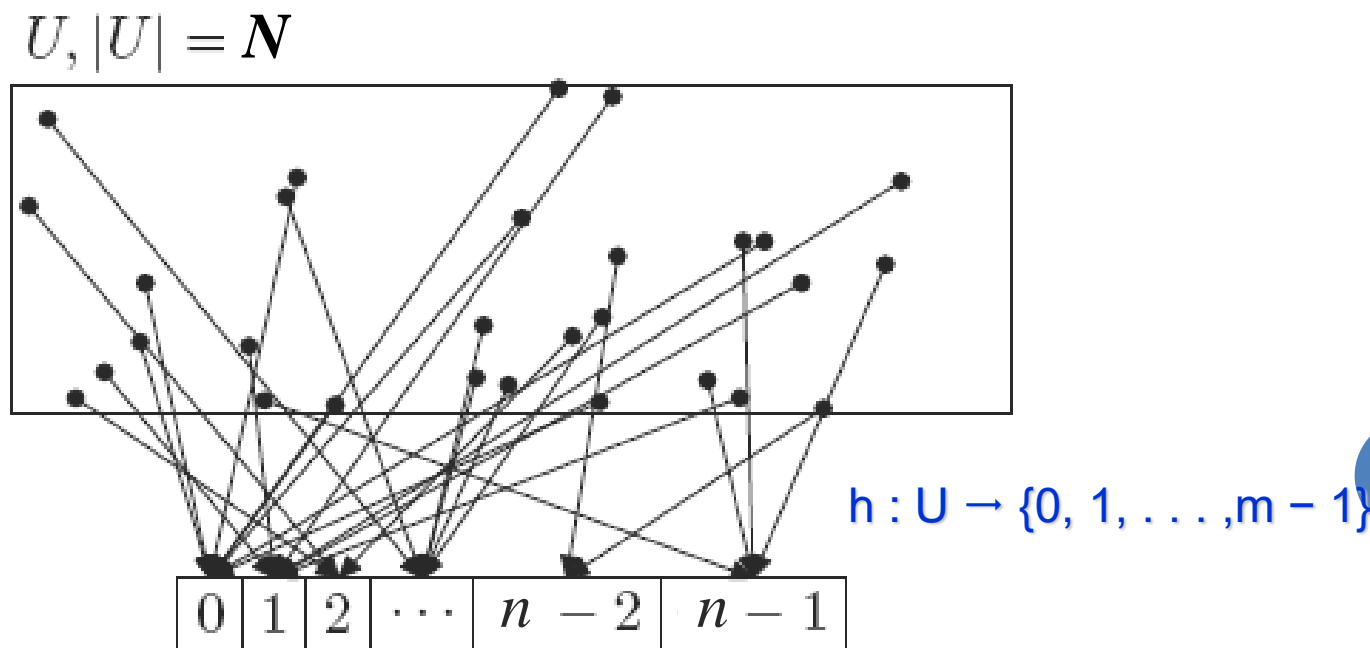
- 14 ?

	IDStud	Ime	BrTel
0			
1	81	Pera	12345
2			
3			
4	64	Sonja	23456
5			
6	36	Mara	34567
7			
8			
9	49	Fića	87654
Indeks	KLJUČ		



KOLIZIJA KOD HASH FUNKCIJE

- **Kolizija**: Ako dva ključa imaju istu vrednost hash funkcije (različiti ključevi preslikavaju se u istu adresu), kažemo da je došlo do **kolizije**.
- Ključevi koji izazivaju koliziju nazivaju se **sinonimi**.



KOLIZIJA KOD HASH FUNKCIJE (2)

- Osnovni problem prilikom izbora *hash* funkcije je:
 - Veliki skup vrednosti ključeva, i
 - Daleko manji broj podataka koji se zaista smeštaju u tablicu.
- Za dimenziju polja gde se smeštaju podaci biramo M , tako da je $M \geq N$.
- Ako je $|K|$ kardinalnost skupa ključeva, tada je $|K| > M$,
- Posledica: *hash* funkcija, označena sa h , nema jednoznačno preslikavanje, odnosno. važi
$$(\exists (i, j) \in K) (i \neq j \Rightarrow h(i) = h(j))$$



HASH FUNKCIJA – DODATNI POJMOVI

- Hash funkcija koja različite ključeve mapira u različite adrese naziva se **perfektna hash funkcija**, za koju važi:

$$(\forall (i, j) \in K) (h (i) = h (j) \Rightarrow i = j)$$

- Perfektna *hash* funkcija **ne generiše kolizije**.
- Mogućnost realizacije?



FAKTOR POPUNJENOSTI TABLICE

- **Faktor popunjenosti** (engl. *load factor*), FP
 - Vađan u procesu projektovanja odnosno **određivanja veličine tablice!!**
 - predstavlja **odnos broja elemenata** u tablici i **broja rezervisanih lokacija** za elemente tablice (veličine tablice):

$$FP = \frac{\text{broj elemenata u tablici}}{\text{veličina tablice}}$$

- “Idealna” vrednost **FP = 0.8**



PREGLED OSNOVNIH POJMOVA

(detaljnije objašnjenje sledi nadalje)

- Hash tablica – struktura podataka
- Hash funkcija – funkcija za preslikavanje ključa u adresu (indeks) tablice
- Ključ – vrednost na osnovu koje se određuje adresa podatka
- Kolizija – problem kada se dva ključa preslikavaju u istu adresu u tablici
- Sinonimi – podaci sa različitim ključevima koji se preslikavaju u istu adresu u tablici
- Faktor popunjenosti – odnos broja elemenata i veličine tablice



IMPLEMENTACIJA HASH TABLICE

- Za Hash tablicu se rezerviše **polje** od **N** ćelija ili elemenata tablice
- Indeksi elemenata su **0,1,2,...,N-1** (ili **1,2, ... , N**).
- **Hash funkcija $h(\text{el_type})$** se implementira kao potprogram koji sukcesvino preslikava skup elemenata određenog tipa na skup celih brojeva između **0** i **N-1**, gde je **N** celobrojna konstanta.
- **Implementacija funkcije dodavanja** elementa u hash tablicu se svodi na to da element **x** smestimo u ćeliju tablice sa indeksom **$h(x)$** .
- **Implementacija funkcije čitanja/trazjenja** je obrnuta – element sa ključem **x** tražimo u ćeliji sa indeksom **$h(x)$** .



OSNOVNE OPERACIJE

- Osnovne operacije za rad sa *hash* tablicama su:
 - **insertItem** – dodaje novi element u tablicu,
 - **find** - pronalazi element tablice sa zadatim ključem
 - **removeElement** - uklanja element sa zadatim ključem iz tablice.
- Dodatne funkcije:
 - **getLength** – vraća veličinu tablice i
 - **getLoadFactor**- vraća faktor popunjenosti tablice.
- Implementacija **primarne** transformacije (*hash* funkcija), i
- Implementacija **sekundarne** transformacije.

OPŠTI POSTUPAK KOD DODAVANJA PODATKA SA KLJUČEM K

Algoritam:

1. Izračunavanje hash funkcije $h(K)$ na osnovu vrednosti ključa;
2. Ako je pozicija $h(K)$ slobodna, podatak se upisuje na to mesto.
3. Inače, lokacija je zauzeta i došlo je do kolizije:
 - 1) vrši se rešavanje kolizije nekom od metoda
 - 2) Menja se vrednost primarne adrese u sekundarne adrese i pokušava upis sve dok se ne nađe slobodna lokacija
 - 3) Ako slobodna lokacija postoji, vrši se upis podatka,
 - 4) Inače, tablica je puna.



OPŠTI POSTUPAK KOD BRISANJA PODATKA SA KLJUČEM K

Algoritam:


1. Izračunavanje hash funkcije $h(K)$ na osnovu vrednosti ključa;
2. Ako je pozicija $h(K)$ zauzeta,
 1. Ako je to podatak sa ključem K, **obrisati ga.**
 2. Inače, nastavi pretragu po sinonimima
 1. Menja se vrednost primarne adrese u sekundarne adrese i pokušava traženje sve dok se ne nađe traženi podatak
 2. Ako podatak postoji, vrši se brisanje podatka,
 3. Inače, podatak koji se traži ne postoji, nemoguće brisanje
3. Inače, podatak sa ključem K ne postoji

KAKO?


obrisati ga.



BRISANJE IZ HASH TABLICE

- Ne može se jednostavno samo obrisati element iz tablice na koji ukazuje $h(k)$. Zašto?
 - Koristi se specijalna vrednost **DELETED** za markiranje ćelije koja je sadržala obrisani element.
 - *Traženje* tretira vrednost DELETED kao da ćelija sadrži element sa ključem koji se ne poklapa sa traženim.
 - *Dodavanje* tretira vrednost DELETED kao da je ćelija prazna tako da se može koristiti za smeštanje novog podatka.
 - **Nedostatak:** Vreme traženja
 - Ulančavanje sinonima kao rešenje ovog problema!
- 

REŠAVANJE KOLIZIJE

- **Kolizija** – dva podatka sa različitim ključevima se preko *hash* funkcije preslikavaju u istu adresu
 - Bez obzira kako izabrali *hash* funkciju, postoji velika verovatnoća da će prilikom unosa podataka doći do kolizije.
 - Kolizije produžavaju vreme pristupa elementima u tablicama, tako da je potrebno što efikasnije rešiti problem smeštanja sinonima.
- 

REŠAVANJE KOLIZIJE (2)


Metode za rešavanje kolizije:

- **Otvoreno adresiranje** – kad dođe do kolizije, polje se pretražuje na izabrani sistematski način sve dok se ne nađe prazna ćelija za upis podatka.
- **Ulančavanje sinonima**
 - **spoljašnje ulančavanje sinonima** – kreira se lančana lista tako da se kod kolizije u nju dodaju sinonimi
 - **unutrašnje ulančavanje sinonima** – ulančavanje se vrši unutar tablice (praktično odgovara definiciji otvorenog adresiranja, s tim da se elementima tablice dodaje još jedno polje – link na sinonime).

OTVORENO ADRESIRANJE

- Kod kolizije traženje se nastavlja traženjem slobodne lokacije korišćenjem **sekundarne transformacije**:

$$h_i(k) = (h(k) + c(i)) \bmod M, \quad i = 0, 1, \dots, M - 1$$

- Funkcija **$c(i)$** naziva se sekundarna transformacija i treba da ima sledeća dva svojstva:
 - **$c(0) = 0$** , čime se obezbeđuje da se prvi pokušaj poklopi sa primarnom (hash) transformacijom, i
 - skup **$\{ c(0) \bmod M, c(1) \bmod M, \dots, c(M-1) \bmod M \}$** mora sadržati sve cele brojeve iz intervala **$[0, M-1]$** , kako bi se obezbedilo adresiranje čitavog adresnog prostora tablice.
- 

OTVORENO ADRESIRANJE

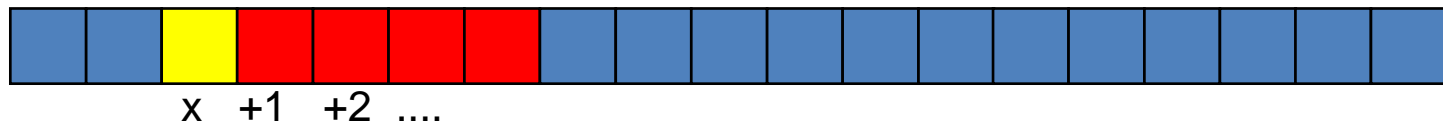
Najčešće korišćene sekundarne transformacije su:

- **Linearno traženje**
- **Modifikovano linearno traženje**
- **Kvadratno traženje**
- **Sekundarna hash funkcija**
- Broj pristupa tablici prilikom čitanja zavisi od popunjenosti tablice.
- Što je tablica punija, to je veći broj koraka potreban da bi se našlo mesto za novi podatak, pa time i broj koraka za njegovo kasnije nalaženje.



LINEARNO TRAŽENJE

- Traženje slobodne ćelije se vrši sekvencijalno (sekvencijalna promena indeksa u tablici) dok se ne dođe do prve slobodne lokacije
 - $c(i) = \alpha \cdot s$, gde je α uzajamno prost broj sa M .
- **Problem:** klasterovanje podataka u nekim delovima tablice
- U tom slučaju, ako je tablica puna, spor pristup
- Treba u proseku manje od 5 pokušaja ako je tablica manje od $2/3$ puna
- U slučaju kada je $\alpha \neq 1$ ova transformacija se naziva **modifikovano linearno traženje**.



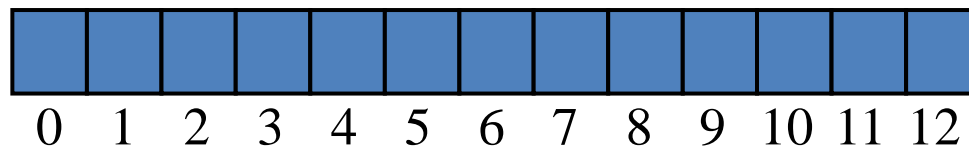
PRIMER LINEARNOG TRAŽENJA

○ Dato je:

- $h(x) = x \bmod 13$
- Treba upisati u navedenom redosledu:
18, 41, 22, 44, 59, 32, 31, 73,

Rešenje:

- Veličina tablice: **13** (**zašto?**)
- Prazna tablica:



PRIMER LINEARNOG TRAŽENJA (2)

$$h(x) = x \bmod 13$$

18, 41, 22, 44, 59, 32, 31, 73,

○ Korak 1:

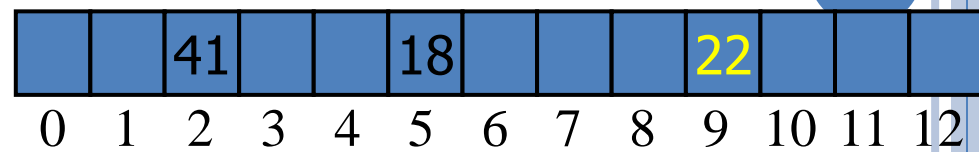
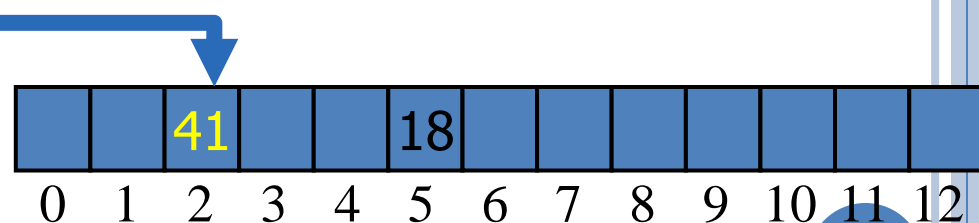
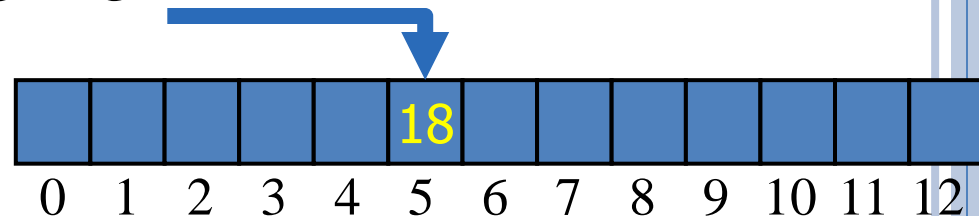
- $k=18$, $h(k)=18 \bmod 13 = 5$

○ Korak 2:

- $k=41$,
- $h(k)=41 \bmod 13 = 2$

○ Korak 3:

- $k=22$,
- $h(k)=22 \bmod 13 = 9$



PRIMER LINEARNOG TRAŽENJA (3)

○ Korak 4:

- $k=44$, $h(k)=44 \bmod 13 = 5 \rightarrow$ **Kolizija**
- $i = h(k) + 1 = 5+1=6 \Rightarrow$ **Slobodna**

		41			18	44			22			
0	1	2	3	4	5	6	7	8	9	10	11	12

○ . . .

○ Posle dodavanja svih elemenata:

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12



TRAŽENJE ELEMENTA SA LINEARNIM TRAŽENJEM MESTA ZA SINONIME

Algoritam HT.1: Traženje u tablici

```
1. find(T,N,k)
2.   i ← h(k)
3.   p ← 0
4.   repeat
5.     c ← T[i]
6.     if c = ∅
7.       return null
8.     else if c.key = k
9.       return T[i]
10.  else
11.    i ← (i + 1) mod N
12.    p ← p + 1
13.  until p = N
14. return null
```

- Traženje u tablici T počinje od lokacije $h(k)$
- Vrš se sukcesivni obilazak lokacija sve dok se ne desi:
 - Našli smo element sa ključem k ,
ILI
 - Našli smo praznu ćeliju,
ILI
 - N ćelija je neuspešno obišeno (N je veličina T)

AŽURIRANJE SA LINEARNIM TRAŽENJEM

Kod operacija dodavanja i brisanja, uveden je specijalni objekat, nazvan **DELETED**, koji zamenjuje obrisane elemente

Algoritam HT.2: Brisanje u tablici

removeElement(T, N, k)

1. Traženje elementa sa ključem k (kao kod **find**)

2. **Ako** je element (k, o) nađen,

Tada zamenjujemo njegovu vrednost specijalnom vrednošću **DELETED** i vraćamo poziciju tog elementa

1. **Inače**, vraća se null pozicija



AŽURIRANJE SA LINEARNIM TRAŽENJEM

Kod operacija dodavanja i brisanja, uveden je specijalni objekat, nazvan **DELETED**, koji zamenjuje obrisane elemente

Algoritam HT.3: Dodavanje

insertItem(T, N, k, o)

1. **Ako** je tabela puna
Tada Prekoračenje **Kraj**
2. Početna ćelija za obradu je $h(k)$
3. Obilazimo konsekvantne ćelije sve dok se ne desi nešto od sledećeg (petlja)
 - Ćelija i je prazna ili sadrži **DELETED**, ili
 - N ćelija je neuspešno obićeno
4. **Ako** je nađena prazna ćelija **tada** Smestimo element (k, o) u ćeliju i

PROBLEM BRISANJA KOD LINEARNOG TRAŽENJA

- $h = \text{key} \bmod 10$
- insertElement:
47, 57, 68, 18, 67
- find: 68
- find: 10
- removeElement: 47
- find: 57

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	



PROBLEM BRISANJA KOD LINEARNOG TRAŽENJA (2)

Pregled:

- Lažno brisanje elemenata – specijalna vrednost DELETED
- Za svaku ćeliju definišemo 3 moguća stanja:
 - aktivna
 - prazna
 - obrisana
- Za operacije **find** ili **removeElement**
 - Pretraživanje se zaustavlja jedino ako je detektovana **prazna** ćelija (ne i za **obrisanu**!!)



PROBLEM BRISANJA KOD LINEARNOG TRAŽENJA (3)

○ InsertElement

- Ćelija prazna ili obrisana
 - Ćelija aktivna
-

insert na poz. H, *cell* = *active*
 $H = (H + 1) \bmod TS$

○ Find

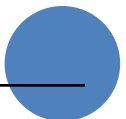
- Ćelija prazna
 - Ćelija obrisana
 - Ćelija aktivna
-

NOT found
 $H = (H + 1) \bmod TS$
key = key in cell FOUND
else $H = (H + 1) \bmod TS$

○ removeElement

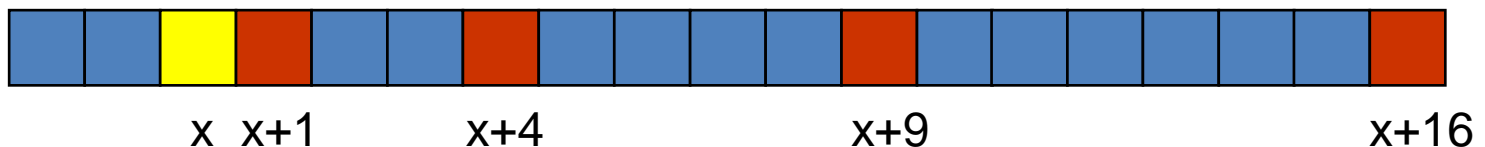
- Ćelija aktivna; key \neq key u ćeliji
 - Ćelija aktivna; key = key u ćeliji
 - Ćelija obrisana
 - Ćelija prazna
-

$H = (H + 1) \bmod TS$
DELETE; *cell* = *deleted*
 $H = (H + 1) \bmod TS$
NOT found



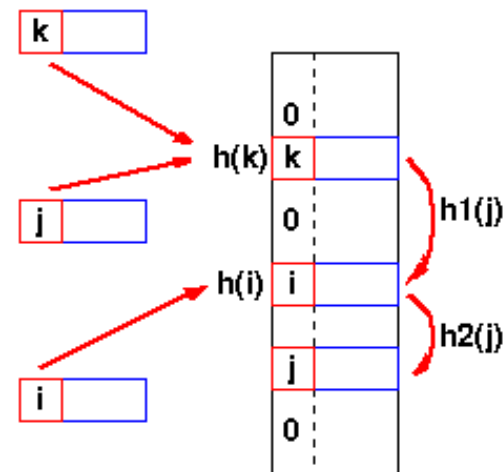
KVADRATNO TRAŽENJE

- Pokušaj da se izbegne problem klasterovanja
- Slobodna lokacija se traži:
 - $c(i) = \alpha \cdot i^2$
- Vrš se distribucija podataka u izdvojene ćelije u tablici
- Ako je indeks x , sledeće ćelije koje se obilaze su $x+1$, $x+4$, $x+9$, $x+16$ itd.
- Eliminiše primarno klasterovanje
- Problem ješto se svi ključevi koji se posle heširanja preslikavaju u istu ćeliju slede istu sekvencu kod traženja slobodne (sekundarno klasterovanje).



SEKUNDARNA HASH FUNKCIJA

- Dvostruko heširanje
- Vrš se transformacija ključa novom hash funkcijom, tzv. Sekundarna hash funkcija, koja se koristi i u svakom narednom koraku za dobijeni ključ:
 - $c(i) = i \cdot h'(k)$, gde je $h'(k)$ *hash* funkcija različita od primarne transformacije.
- Bolje rešenje u odnosu na prethodna, manji broj pokušaja u odnosu na linearno traženje
- Generiše sekvence koje zavise od novo generisanog ključa, i nisu iste za svaki ključ.
- Promena koraka je konstantna, ali se razlikuje za različite ključeve.
- Ne sme nikad da vrati 0.
- Zahteva da veličina tablice bude prost broj.



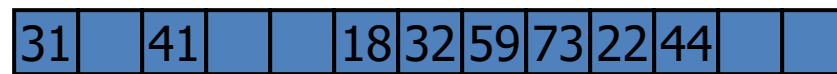
PRIMER DVOSTRUKOG HEŠIRANJA

- Hash tablica sadrži cele brojeve
- Kolizija – dvostruko heširanje
 - $N = 13$
 - $h(k) = k \bmod 13$
 - $d(k) = 7 - k \bmod 7$
- Dodati:
18, 41, 22, 44, 59,
32, 31, 73, u
navedenom
redosledu

k	$h(k)$	$d(k)$	Adresa	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	
31	5	4	5	9 0
73	8	4	8	



0 1 2 3 4 5 6 7 8 9 10 11 12



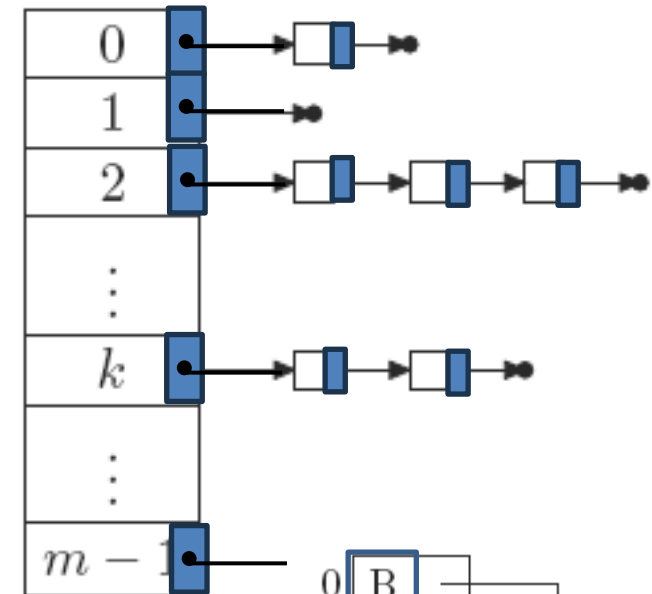
0 1 2 3 4 5 6 7 8 9 10 11 12

REŠAVANJE KOLIZIJE – ILUSTRACIJA ULANČAVANJA SINONIMA

○ Ulančavanje sinonima

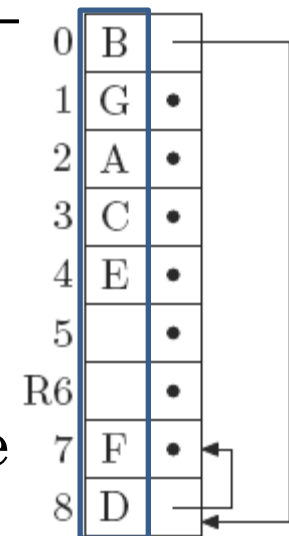
- **Spoljno ulančavanje**

- Svaki element tablice sadrži element, ukazatelj na početak l. liste koja sadrži sinonime



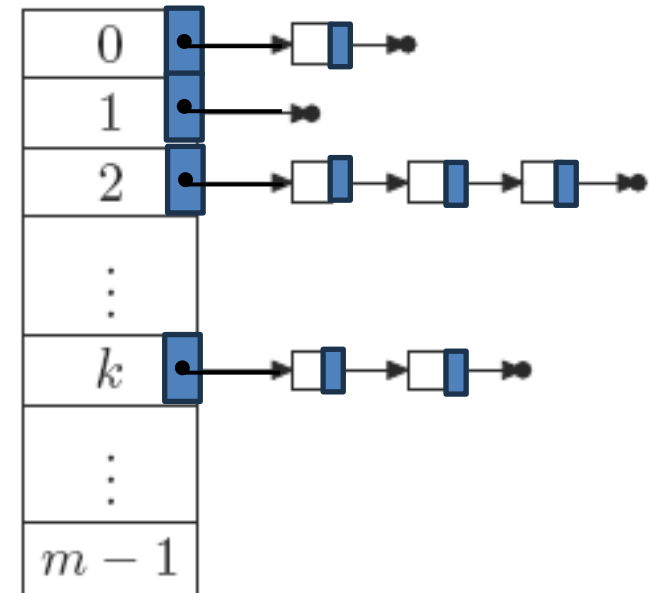
- **Unutrašnje ulančavanje**

- implementirano po ugledu na statičku implementaciju lančanih listi
- 2D polje, gde je jedna kolona za podatke a druga kolona za „link“ unutar tablice



SPOLJAŠNJE ULANČAVANJE SINONIMA

- *Hash* tablica je organizovana kao **vektor lančanih listi**.
- *Hash* funkcija određuje u kojoj od lančanih listi se nalazi traženi podatak
- Lista obilazi standardnim metodom praćenja linkova.
- Broj elemenata liste je broj sinonima M
- Taj broj može da utiče na brzinu pristupa



SPOLJAŠNJE ULANČAVANJE

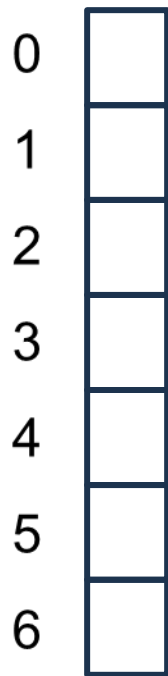
SINONIMA

- Redukuje se broj poređenja za faktor M , ali se koristi dodatni memorijski prostor za M linkova
- **Brisanje elemenata** iz tablice ne predstavlja problem – brisanje iz lančane liste
- Veličina tablice ne mora biti prost broj
- Mogu se koristiti i polja tzv *buckets* na svakoj lokaciji u hash tablici umesto ulančavanja.

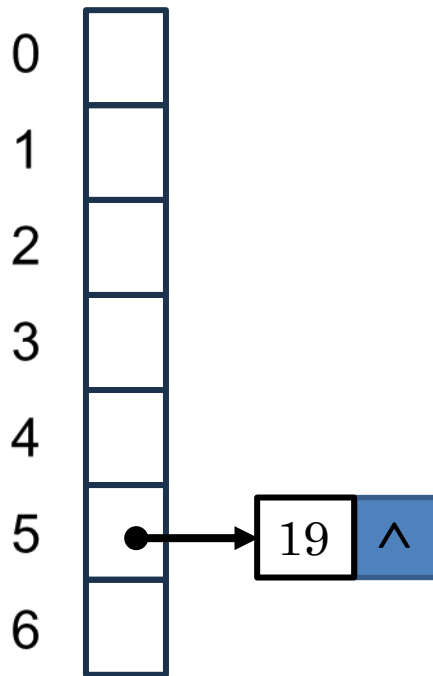


SPOLJAŠNJE ULANČAVANJE SINONIMA

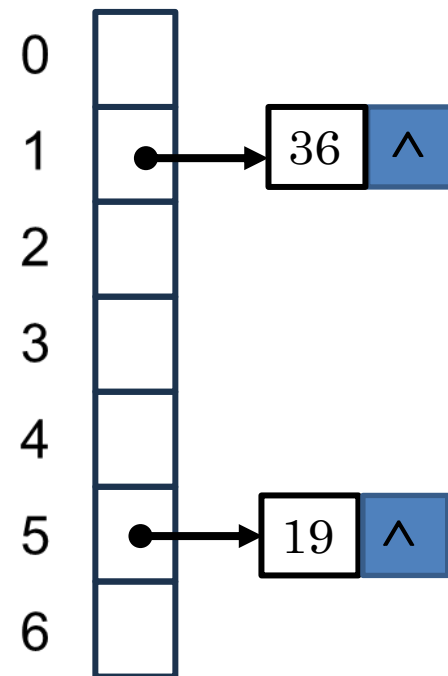
- Primer: $N = 7$, $h(k) = |k| \bmod N$
- Dodati elemente: 19 36 5 21 -4 26 14



○ Početno stanje



Dodavanje 19

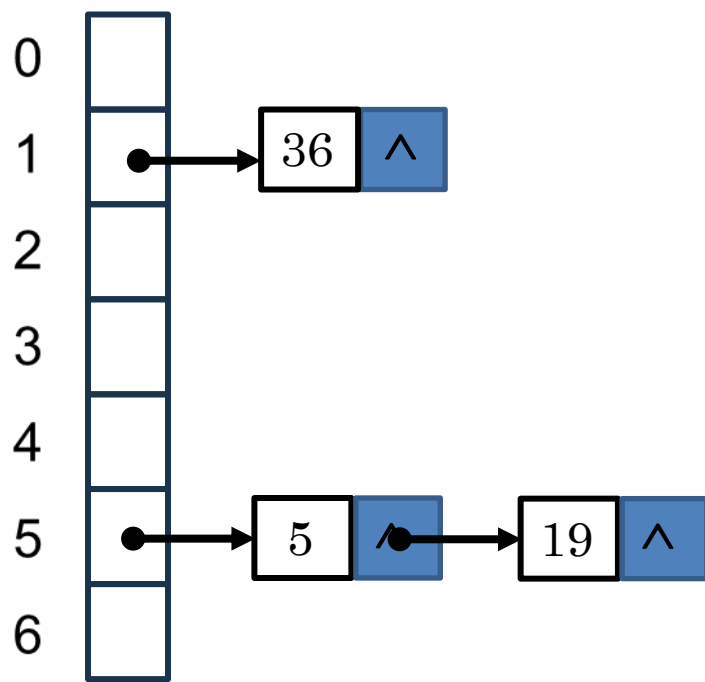


Dodavanje 36

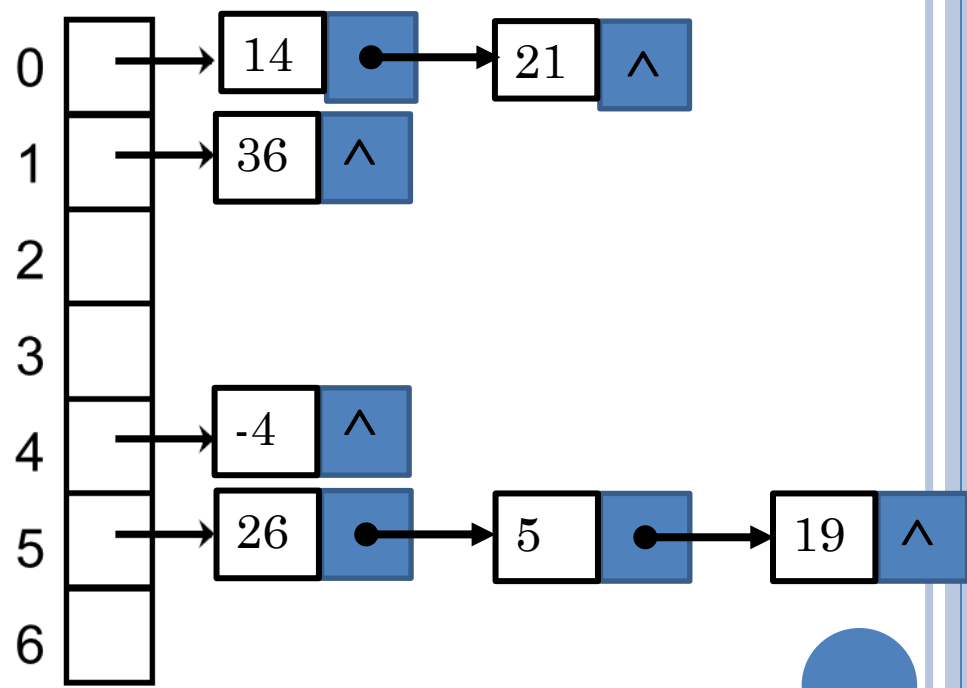


SPOLJAŠNJE ULANČAVANJE SINONIMA

- Primer: $N = 7$, $h(k) = |k| \bmod N$
- Dodati elemente: 19 36 5 21 -4 26 14



• Dodavanje 5 ...



• konačni izgled Heš tablice

UNUTRAŠNJE ULANČAVANJE SINONIMA

- Svaka vrsta u tablici mora sadržati bar dva polja:
 - ključ
 - pokazivač na sledeći sinonim.
- *Hash* funkcijom određuje se pozicija prvog sinonima, a ukoliko traženi podatak nije na toj poziciji, traženje se nastavlja sleđenjem “linkova”
- Sinonimi mogu biti postavljeni bilo gde u okviru tablice

0	B	—
1	G	•
2	A	•
3	C	•
4	E	•
5		•
R6		•
7	F	•
8	D	—

UNUTRAŠNJE ULANČAVANJE SINONIMA

- Upis se vrši u **dva prolaza**
 - Sinonimi se smeju postavljati tek nakon zauzimanja svih lokacija dobijenih transformacijama ključa
 - Smeštanje se najčešće obavlja linearnim traženjem slobodne lokacije, počev od pozicije koju generiše *hash* funkcija. Kasnije dodavanje novih podataka u tablicu zahteva restrukturiranje tablice.
- Za sinonime se može rezervisati poseban prostor unutar tablice.



OSObine HASH TABLICA

- Struktura podataka koja obezbeđuje brz pristup i dodavanje podataka, *uglavnom* reda **$O(1)$** .
- Relativno se **lako programira** (u odnosu na druge složene strukture kao što su stabla ili grafovi napr.)
- **Implementira** se preko **polja**, s tim da pozicija elementa u polju zavisi od aritmetičke transformacije ključa.
- Pošto se implementacija zasniva na poljima, postoji poznati **problem ekspanzije polja**.
- **Ne postoji** podesan način za **obilazak elemenata** hash tablice u bilo kom redosledu.



EFIKASNOST HASHING-A

- Dodavanje i traženje je reda $O(1)$.
- Kod **kolizije**, vreme pristupa zavisi od dužine traženja mesta za sinonime.
- Pojedinačna operacija upisa ima **složenost koja je proporcionalna dužini traženja** mesta za sinonime.
- Kvadratno traženje i dvostruko heširanje su istih performansi.
 - Kod uspešnog traženja: $\log_2(1 - \text{loadFactor}) / \text{loadFactor}$
 - Neuspešno traženje: $1 / (1 - \text{loadFactor})$
- Traženje sa spoljašnjim ulančavanjem: $1 + \text{loadFactor} / 2$
 - Kod uspešnog traženja: $1 + \text{loadFactor}$
 - Kod umetanja: $1 + \text{loadfactor}^2$ za uređene liste i 1 za neuređene.
- Ako se koristi otvoreno adresiranje, dvostruko heširanje ima prednost u odnosu na kvadratno traženje.
- Ako imate na raspolaganju veliku količinu memorije, i ne zahteva se povećanje količine podataka, linearno traženje je vrlo jednostavno za implementaciju.
- Ako je količina podataka nepoznata, spoljašnje ulančavanje ima prednost
- Ako ste u dilemi šta da koristite, implementirajte spoljašnje ulančavanje



PITANJA, IDEJE, KOMENTARI

