Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Slovar

razpršene tabele in Bloomov filter

Razpršena tabela

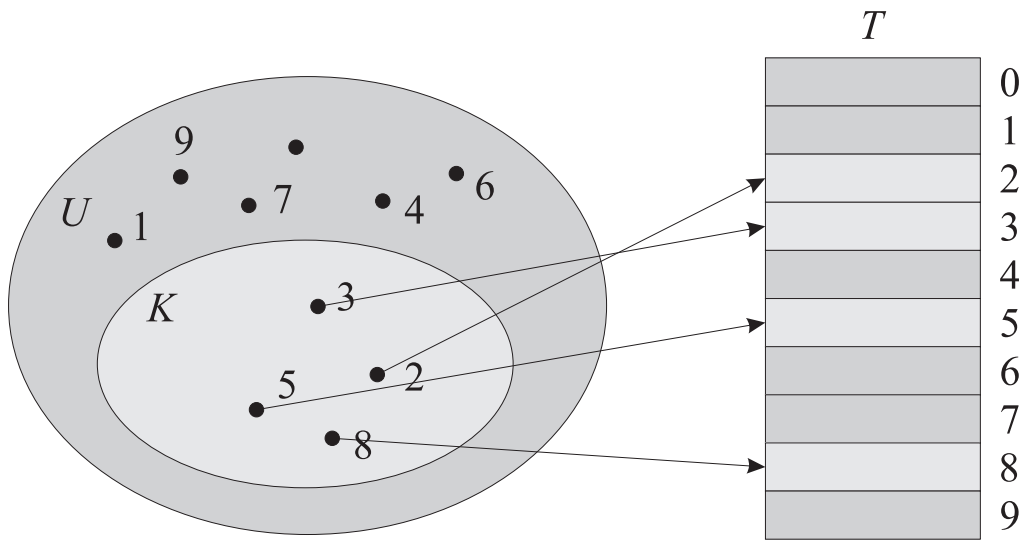
Lahko jo tudi imenujemo zgoščena tabela.

- Ključi, s katerimi imamo opravka, so iz neke univerzalne množice U. Recimo, da je število predmetov, ki jih hranimo v slovarju n.
- Naredimo polje (tabelo) predmetov `S` iz razreda `Elt` tako, da se predmet `elt` nahaja na mestu `S[elt.key]`.
- Če je $|U|$ velika, in je n glede na $|U|$ majhno število, je ta način prostorsko **potraten**.
- Rešitev: tabela `S` naj bo $\gg$primerno velika$\ll$, manjša od $|U|$.
- Mesto v tabeli `S` dolžine m, kamor vstavimo ključ, določa *funkcija zgoščanja* h

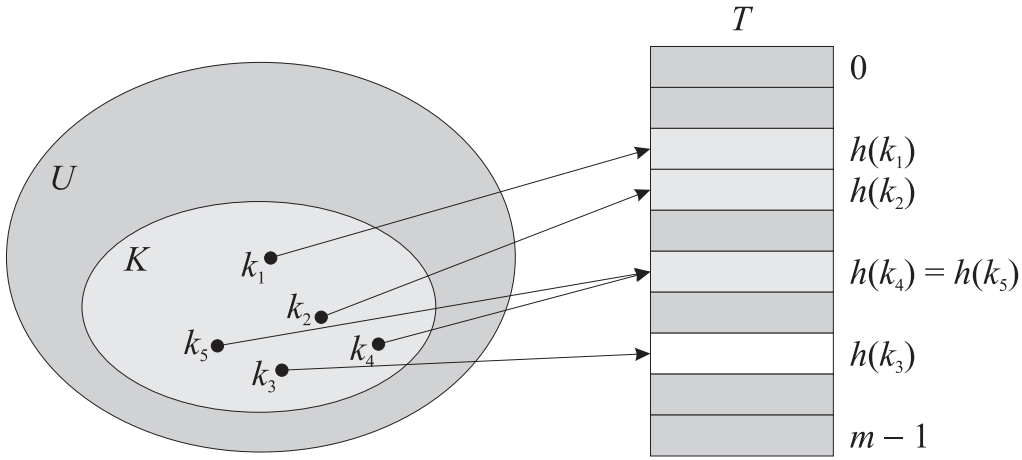
$$h : U \to \{0, 1, 2, \ldots, m-1\}$$

Razpršena tabela – nadalj.

- Ključ `key` (točneje podatek `elt` s ključem `elt.key`) se nahaja na mestu `S[h(elt.key)]`.

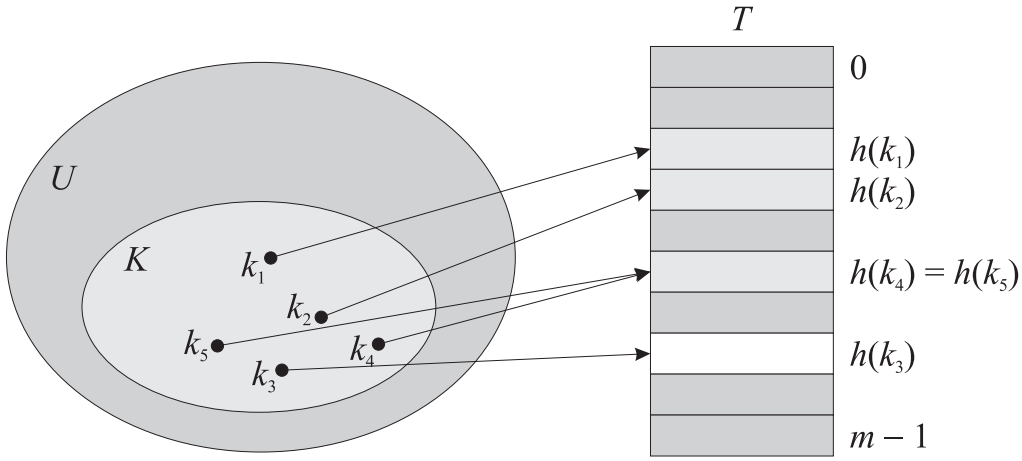


Sovpadanje



- Situacijo (*težavo*), ko je $h(k_4) = h(k_5)$ imenujemo *kolizija* ali *sovpadanje* – dva predmeta bi morala biti na istem mestu v tabeli `S`.
- Sovpadanjem se ne moremo izogniti, saj predpostavljamo, da je $m \ll |U|$. (h ne more biti injektivna funkcija.)
- Dobro je izbrati h, ki minimizira število sovpadanj (konstantna funkcija gotovo ni primerna), a hkrati zagotavlja velikost tabele $m = O(n)$.

Sovpadanje



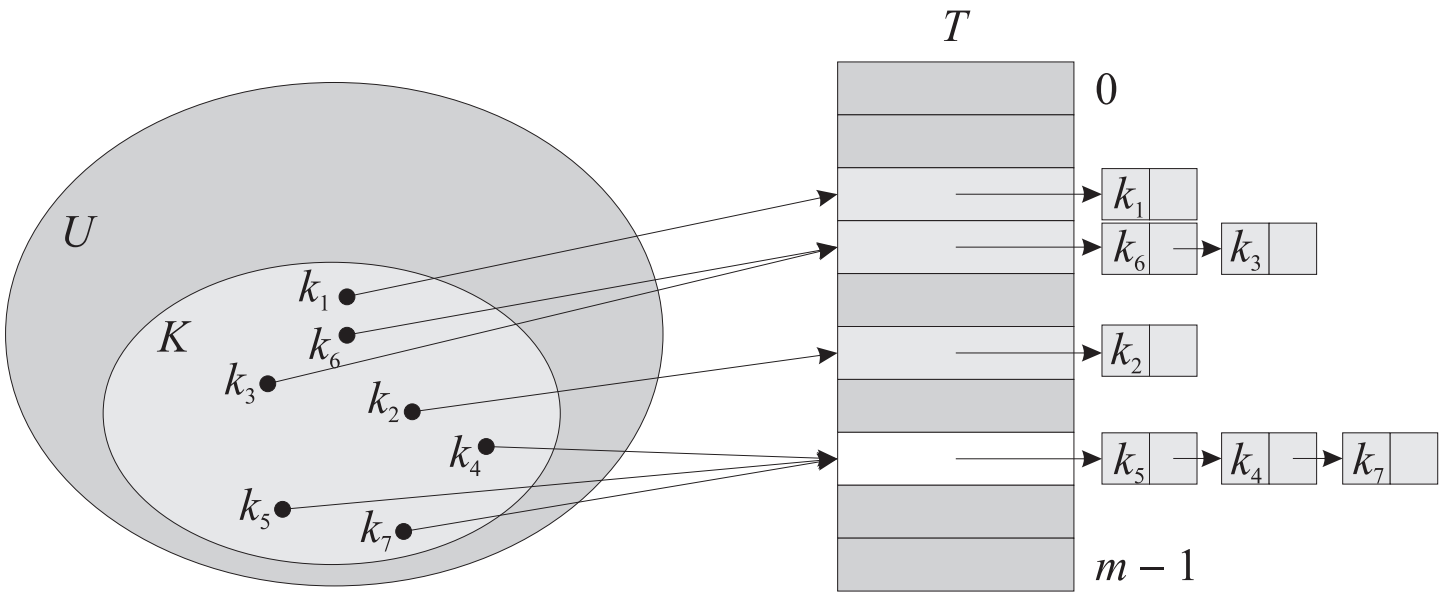
- Situacijo (*težavo*), ko je $h(k_4) = h(k_5)$ imenujemo *kolizija* ali *sovpadanje* – dva predmeta bi morala biti na istem mestu v tabeli `S`.
- Sovpadanjem se ne moremo izogniti, saj predpostavljamo, da je $m \ll |U|$. (h ne more biti injektivna funkcija.)
- Dobro je izbrati h, ki minimizira število sovpadanj (konstantna funkcija gotovo ni primerna), a hkrati zagotavlja velikost tabele $m = O(n)$.

Reševanje sovpadanja:

- veriženje ali
- naslavljanje

Veriženje

Veriženju se v angleščini reče *chaining*.
Vse elemente, ki se preslikajo na isto mesto v tabeli, hranimo v seznamu.



Veriženje – implementacija operacij

```
public class HTchaining implements Slovar {
  Seznam[] tabela;
  ...
  private int Hash(int v) { ...  }
  public void Insert(Elt x) {
    int i= Hash(x.key);
    tabela[i].Insert= tabela[i].Insert(x);
  }
  public Object Find(int key) {
    int i= Hash(x.key);
    return tabela[i].Find(key);
  }
  public nekajDrugega Delete(int key) {
    // za domačo nalogo
  }
}
```

Veriženje – zahtevnost

Časovna zahtevnost iskanja v tabeli z n shranjenimi elementi.

- V najslabšem primeru $\Theta(n)$ (vsi elementi se preslikajo na isto mesto v tabeli in moramo preiskati celoten seznam).
- Če je povprečno število ključev, ki se preslikajo v isto polje tabele α, potem je časovna zahtevnost v povprečju $\Theta(\alpha)$.
- Želimo si $\alpha = O(1)$. To je odvisno od zgoščevalne funkcije *in* od podatkov.

Razpršilna funkcija

Kakšna je dobra funkcija zgoščanja?

- Za vsak ključ k je enako verjetno, da se preslika na katerokoli mesto tabele.
- Bolj natančno: naj bo $P(k)$ verjetnost, da izberemo ključ k. Potem

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{za } j = 0, 1, \ldots, m-1.$$

- Primer: Naj bodo ključi k naključna realna števila enakomerno porazdeljena na intervalu $[0, 1)$. Potem funkcija

$$h(k) = \lfloor k \cdot m \rfloor$$

zadošča zgornjemu pogoju. Funkcija *razpršuje*.

Zgoščevalna funkcija – metoda deljenja

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

- Primer: če je $m = 12$ in je $k = 100$, potem je $h(k) = 4$.
- Odlika: hitrost. (Komentar?)
- Na kaj moramo paziti: izogibati se moramo nekaterim vrednostim m. Na primer: ni dobro, če je m potenca števila 2, to je če je $m = 2^p$, potem je $h(k)$ odvisna le od p bitov ključa k. Je pa to hitra operacija: pomik in bitni in.
- Dobre vrednosti m so praštevila, ki niso blizu potence 2.
- Primer: če želimo shraniti približno 2000 ključev in je za dolžino seznamov pri veriženju sprejemljivo število 3, potem izberemo za m število 701. To je praštevilo, ki ni blizu nobeni potenci števila 2 in je blizu $2000/3$.
- Se pa lahko zalomi. Na primer, ko so ključi oblike m^k.
- Žal porazdelitve običajno ne poznamo.
- Na pomoč: *paradoks rojstnega dne*. Kaj je to? Kako deluje? Zakaj tako deluje?

Zgoščevalna funkcija – metoda množenja

$$h(k) = (k \cdot p) \bmod m \ ,$$

kjer je p neka konstanta.

- Vrednost m tu ni kritična.
- Kaj je z vrednostjo p? Izkaže se, da je najbolje, če je p praštevilo.
- $h()$ lahko zapišemo tudi kot:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor \ ,$$

kjer $0 < A < 1$ in $x \bmod 1$ pomeni decimalni del x. V tem primeru Knuth $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$.

Naslavljanje

- Za shranjevanje podatkov sedaj uporabljamo samo polja tabele.
- V primeru sovpadanja izračunamo nov indeks tabele, kamor bomo vstavili element. Če je tudi to mesto že zasedeno, postopek ponavljamo, dokler ne najdemo prostega mesta (če tabela ni že polna).
- Problem: kako naračunati zaporedje indeksov (poskusov) tako, da bomo
  - uporabili čim manj poskusov preden bomo našli prosto mesto,
  - poskusili vstaviti v vsa polja tabele.
- Slabosti:
  - omejen prostor
  - težava pri brisanju
- Angleški izraz: *open addressing*.

Naslavljanje

Formalno zgoščevalna funkcija sedaj slika

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\},$$

to pomeni, da bomo najprej poskusili vstaviti element s ključem k v polje $h(k, 0)$, nato (če je to polje že zasedeno) v $h(k, 1)$, nato v $h(k, 2)$, ...
Da je funkcija sedaj dobra, zanjo veljajo:

- pogoji iz prosojnice 9 – verjetnost slikanja v vsako polje tabele je (približno) enaka; in
- za vsak k mora biti *zaporedje poskusov*

  $$(h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1))$$

  permutacija zaporedja $(0, 1, 2, \ldots, m-1)$. To pomeni da za vsak ključ preizkusimo vsako polje tabele.

Naslavljanje – implementacija operacij

```
public class HTopen implements Slovar {
  Seznam[] tabela;
  ...
  private int Hash(int v, int j) { ...  }
  public void Insert(Elt x) {
    for (int j= 0; tabela[ Hash(x.key, j) ] != NULL; j++)
    tabela[Hash(x.key, j)]= x;
  }
  public Object Find(int key) { ...  }
  public nekajDrugega Delete(int key) { ...  }
}
```

Naslavljanje – implementacija operacij

```
public class HTopen implements Slovar {
  Seznam[] tabela;
  ...
  private int Hash(int v, int j) { ...  }
  public void Insert(Elt x) {
    for (int j= 0; tabela[ Hash(x.key, j) ] != NULL; j++)
    tabela[Hash(x.key, j)]= x;
  }
  public Object Find(int key) { ...  }
  public nekajDrugega Delete(int key) { ...  }
}
```

Kaj če je tabela polna?

Naslavljanje – implementacija operacij

```
public class HTopen implements Slovar {
  Seznam[] tabela;
  ...
  private int Hash(int v, int j) { ...  }
  public void Insert(Elt x) {
    for (int j= 0; tabela[ Hash(x.key, j) ] != NULL; j++)
    tabela[Hash(x.key, j)]= x;
  }
  public Object Find(int key) { ...  }
  public nekajDrugega Delete(int key) { ...  }
}
```

Kaj če je tabela polna?

- Lahko ne dovolimo vstavljanja – `exception`.
- Lahko naredimo novo tabelo dvojne velikosti in vanjo prestavimo vse elemente iz stare tabele – *doubling*.
- Kako izgleda $h(\_, i)$?

Linearno naslavljanje

- Naj bo $h' = h(k, 0)$ in $h_i = h(k, i)$, kjer je $i > 0$. Pri linearnem naslavljanju potem velja

$$h(k, i) = (h_{i-1} + 1) \bmod m = (h'(k) + i) \bmod m$$

  V resnici ni nujno, da prištejemo 1, ampak lahko prištejemo poljubno konstanto c.
- Slabost je, da se lahko tvorijo se zaporedja polnih polj, kar podaljšuje povprečni čas iskanja prostega polja.
- Če je polje prosto in je pred njim že i polnih polj, potem je verjetnost, da bomo to polje zapolnili, enaka $(i + 1)/m$, če pa je polje pred tem poljem prazno, je verjetnost, da ga zasedemo, $1/m$.
- Če imamo prosto vsako sodo polje in je vsako liho polje zasedeno, potem povprečno potrebujemo $1,5$ poskusa.
- Če je zasedenih prvih $m/2$ polj tabele, potem povprečno potrebujemo že $m/8$ poskusov.

Kvadratično naslavljanje

- Naj bo $h' = h(x, 0)$, in konstanti c_1 in $c_2 \neq 0$. Potem pri kvadratičnem naslavljanju velja:

$$h(k, i) = (h'(k) + c_1 i^2 + c_2 i) \bmod m$$

- S kvadratičnim naslavljanjem smo se znebili zaporedij sovpadanj.
- Toda, če se dva ključa s h' preslikata v isto vrednost, potem se zaporedje sovpadanj ohranja.
- Imamo $\Theta(m)$ možnih zaporedij.

Kvadratično naslavljanje malce drugače

Spet imamo funkcijo zgoščanja h', ki slika iz množice ključev v množico $\{0, 1, \ldots, m-1\}$, kjer $m = 2^k$. POZOR: slednja predpostavka je povsem običajna. Zakaj?
Postopek iskanja naj bo naslednji.

```
public Elt Find(int key) {
  for (int i= h'(key), int j= 0;
       (tabela[i].key != key) &&
       (tabela[i].key != NULL);
       j= (j+1) % m, i= (i+j) % m);
  return tabela[i];
}
```

- Algoritem je poseben primer kvadratičnega naslavljanja. Kakšni sta konstanti c_1, c_2?
- Algoritem v najslabšem primeru preišče vsako polje v tabeli.

Dvojno naslavljanje (*double hashing*)

Problem sovpadanja smo reševali:

- z linearno funkcijo: $(h'(k) + \mathbf{ci}) \bmod m$
- s kvadratično funkcijo: $(h'(k) + \mathbf{c_1 i} + \mathbf{c_2 i^2}) \bmod m$
- v splošnem je lahko poljubna funkcija: $(h'(k) + \mathbf{i}h''(\mathbf{k})) \bmod m$ in temu rečemo *dvojno naslavljanje*.
- Ker je sedaj h odvisna od dveh načinov zgoščanja, od h' in h'', imamo $\Theta(m^2)$ možnih zaporedij.
- To odpravi težavo s prosojnice 18, če se dva ključa s h' preslikata v isto vrednost (potem se zaporedje sovpadanj ohranja).
- Na kaj moramo paziti? Vrednosti $h''(k)$ morajo biti za vsak k tuje proti m, sicer, če je $d = \gcd(h''(k), m) > 1$, preiščemo le $(1/d)$-tino tabele. Možni rešitvi
  - m je praštevilo.
  - $m = 2^p$ in poskrbimo, da je $h''(k)$ vedno liho število.

Analiza zgoščanja z naslavljanjem

Imamo $m!$ permutacij indeksov tabele in vsaka permutacija predstavlja niz vrednosti, ki jih vrača naša zgoščevalna funkcija.
Recimo, da je pri vsakem ključu k verjetnost, da dobimo enega izmed $m!$ zaporedij, enaka ter da je enako verjetno, da iščemo katerikoli ključ.
Naj bo *faktor napolnitve tabele* $\alpha = n/m < 1$.
Potem je pričakovano število poskusov največ

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}.$$

Npr., če je tabela napol polna ($\alpha = 1/2$), potem bomo pričakovano število poskusov manjše od $3,4$. Če je tabela 90% polna, bomo pričakovano potrebovali manj kot $3,7$ poskusa.
V obeh primerih je čas dostopa $O(1)$. Kaj manjka pri tej izjavi?

Zahtevnost

	`Find`	`Insert`	`Delete`
seznam	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$O(n)$	$O(n)$	$O(n)$
binarno drevo	$O(n)$	$O(n)$	$O(n)$
AVL drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
B drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
RB-drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
preskočna vrsta	$O(\log n)$	$O(\log n)$	$O(\log n)$
razpršena tabela	$O(1)$	$O(1)$	$O(1)$

Zahtevnost

	`Find`	`Insert`	`Delete`
seznam	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$O(n)$	$O(n)$	$O(n)$
binarno drevo	$O(n)$	$O(n)$	$O(n)$
AVL drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
B drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
RB-drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
preskočna vrsta	$O(\log n)$	$O(\log n)$	$O(\log n)$
razpršena tabela	$O(1)$	$O(1)$	$O(1)$

- Kakšen je čas pri razpršeni tabeli: največji, najmanjši, povprečni, pričakovan?
- Ne dâ se narediti največji (Dietzfelbinger in ostali):
  Za slovar velja $\Omega(\log n)$, *če je prostor* $O(n)$.

Zahtevnost

	`Find`	`Insert`	`Delete`
seznam	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$O(n)$	$O(n)$	$O(n)$
binarno drevo	$O(n)$	$O(n)$	$O(n)$
AVL drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
B drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
RB-drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
preskočna vrsta	$O(\log n)$	$O(\log n)$	$O(\log n)$
razpršena tabela	$O(1)$	$O(1)$	$O(1)$

- Kakšen je čas pri razpršeni tabeli: največji, najmanjši, povprečni, pričakovan?
- Ne dâ se narediti največji (Dietzfelbinger in ostali):
  Za slovar velja $\Omega(\log n)$*, če je prostor* $O(n)$*.*
- Kaj pa, če poskusimo zmanjšati prostor? S čem bomo plačali?

Nenatančni slovar

Imamo slovar S nekakšnih elementov. S tem slovarjem želimo početi vsaj naslednje operacije:

iskanje: `Find(`S`, x) --> y` – v slovarju S poiščemo element `x`. Rezultat `y` je Boolova vrednost `true` ali `false`. Želimo si, da je odgovor praviloma pravilen – dovolimo občasno lažne pozitivne odgovor *false positive*.

Rešitev – perfektna razpršilna funkcija (*perfect hashing function*).

Nenatančni slovar

Imamo slovar S nekakšnih elementov. S tem slovarjem želimo početi vsaj naslednje operacije:

iskanje: `Find(`S`, x) --> y` – v slovarju S poiščemo element `x`. Rezultat `y` je Boolova vrednost `true` ali `false`. Želimo si, da je odgovor praviloma pravilen – dovolimo občasno lažne pozitivne odgovor *false positive*.

Rešitev – perfektna razpršilna funkcija (*perfect hashing function*).
Morda želimo še početi:

dodajanje: `Insert(`S`, x)` – v slovar S dodamo nov element `x`.

Nenatančni slovar

Imamo slovar S nekakšnih elementov. S tem slovarjem želimo početi vsaj naslednje operacije:

iskanje: `Find(`S`, x) --> y` – v slovarju S poiščemo element `x`. Rezultat `y` je Boolova vrednost `true` ali `false`. Želimo si, da je odgovor praviloma pravilen – dovolimo občasno lažne pozitivne odgovor *false positive*.

Rešitev – perfektna razpršilna funkcija (*perfect hashing function*).
Morda želimo še početi:

dodajanje: `Insert(`S`, x)` – v slovar S dodamo nov element `x`.

Odpovemo pa se (za sedaj):

izločanje: `Delete(`S`, x) --> y` – iz slovarja S izločimo element `x`. Rezultat `y` je lahko Boolova vrednost `true` ali `false`, ki sporoči ali je bil element uspešno izločen ali ne, ali pa operacija ničesar ne vrne.

Nenatančni slovar – s štetjem

Torej imamo operaciji:

dodajanje: `Insert(`S`, x)` – v slovar S dodamo element `x` – ni nujno nov!

iskanje: `Find(`S`, x) --> y` – v slovarju S poiščemo element `x`. Rezultat `y` je število pojavitev elementa v slovarju.

Še vedno se odpovemo (za sedaj):

izločanje: `Delete(`S`, x) --> y` – iz slovarja S izločimo element `x`. Rezultat `y` je lahko Boolova vrednost `true` ali `false`, ki sporoči ali je bil element uspešno izločen ali ne, ali pa operacija ničesar ne vrne.

Literatura

Primer:
Andrei Broder, Michael Mitzenmacher: *Network Applications of Bloom Filters: A Survey*, Internet Mathematics, Vol. 1, No. 4, 485-509.

Primer

Imamo slovar n-teric v DNK in nas zanima, koliko je katerih n-teric v določenem DNK.
Imejmo naslednji DNK:

`TAACCCT ...`

Potem imamo naslednje število pojavitev 3-teric v njej:

`AAC` : 1

`ACC` : 1

`CCC` : 2

`CCT` : 1

`TAA` : 1

Prostorska in časovna zahtevnost čim manjša.

Nenatančni slovar – naivna izvedba

Uporabimo dosedanje znanje.

	prostor	`Find`	`Insert`	`Delete`
seznam	$n + rn$	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$n + rn$	$O(n)$	$O(n)$	$O(n)$
binarno drevo	$n + 2rn$	$O(n)$	$O(n)$	$O(n)$
AVL drevo	$n + 2rn$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
B drevo	$n + brn$	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
RB-drevo	$n + 2rn$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
preskočna vrsta	$n+?rn$	$O(\log n)$	$O(\log n)$	$O(\log n)$
razpršena tabela	$n+?rn$	$O(1)$	$O(1)$	$O(1)$

Ali lahko naredimo operacije v času $O(1)$ in prostoru $O(n) = cn$ bitov prostora za nek majhen c (na primer $c = 5$)?
Primerjava: binarno drevo potrebuje $(n + 2rn) \lg n$ (ali celo $64(n + 2rn)$) bitov prostora – delo z velikimi količinami podatkov *big data*.

Bloomov filter – izvedba 1

Podatkovna struktura:

- imamo bitno polje `BF[0..m-1]`, kjer je $m = cn$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`:

```
Insert(S, x):
  za vsak i= 1..k:
    naračunaj li= hi(x);
    BF[li] = 1
```

Bloomov filter – izvedba 1

Podatkovna struktura:

- imamo bitno polje `BF[0..m-1]`, kjer je $m = cn$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`:

```
Insert(S, x):
  za vsak i= 1..k:
    naračunaj li= hi(x);
    BF[li] = 1
```

iskanje: `Find(`S`, x)`:

```
Find(S, x):
  za vsak i= 1..k naračunaj li= hi(x);
  če so vsi BF[li] == 1 ==> return true
  sicer return false
```

Časovna zahtevnost: $O(k)$, prostorska zahtevnost: $O(m) = O(n)$ bitov.

Bloomov filter – izvedba 2

Podatkovna struktura:

- imamo k bitnih polj `BFi[0..m-1]`, kjer je $m = \frac{cn}{k}$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`: podobno kot prej

iskanje: `Find(`S`, x)`: podobno kot prej

Časovna zahtevnost: $O(k)$, prostorska zahtevnost: $O(km) = O(n)$ bitov. Preprostješa izvedba, ki ima enake lastnosti in zato pogosteje uporabljana. Omogoča preprosto povzporejanje.

Kako je z lažnimi pozitivnimi odgovori?

Verjetnost lažnega pozitivnega odgovora

Recimo, da imamo prvo izvedbo, ker jo bo lažje analizirati. Rezultati so podobni za drugo.
Ker imamo dobre razpršilne funkcije, je verjetnost, da je nek bit 0 po vstavljanju n elementov

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} = p$$

potem je *pričakovana* vrednost, da dobimo lažen pozitiven odgovor, kar pomeni, da dobimo k enic, približno enaka

$$f \approx (1-p)^k = \left(1 - e^{\frac{-kn}{m}}\right)^k \quad .$$

Izziv: Analizirajte zgornje vrednosti za različne m, c, n in k.

Razmisleki

Smo pri prvi izvedbi.

- če povečamo k, se bo (intuitivno) zmanjšala verjetnost, da bomo našli bit 0 pri razprševanju;
- če zmanjšamo k, se bo zmanjšalo število bitov 0, kar pomeni, da bo f manjši.

Razmisleki

Smo pri prvi izvedbi.

- če povečamo k, se bo (intuitivno) zmanjšala verjetnost, da bomo našli bit 0 pri razprševanju;
- če zmanjšamo k, se bo zmanjšalo število bitov 0, kar pomeni, da bo f manjši.

Optimizacija? Matematika na pomoč!!

Razmisleki

Smo pri prvi izvedbi.

- če povečamo k, se bo (intuitivno) zmanjšala verjetnost, da bomo našli bit 0 pri razprševanju;
- če zmanjšamo k, se bo zmanjšalo število bitov 0, kar pomeni, da bo f manjši.

Optimizacija? Matematika na pomoč!!
Naj bo $g = \ln f = k \ln(1 - p)$ potem je optimum tedaj, ko je parcialni odvod $\frac{dg}{dk} = 0$.

Razmisleki

Smo pri prvi izvedbi.

- če povečamo k, se bo (intuitivno) zmanjšala verjetnost, da bomo našli bit 0 pri razprševanju;
- če zmanjšamo k, se bo zmanjšalo število bitov 0, kar pomeni, da bo f manjši.

Optimizacija? Matematika na pomoč!!
Naj bo $g = \ln f = k \ln(1 - p)$ potem je optimum tedaj, ko je parcialni odvod $\frac{dg}{dk} = 0$.

Ostalo prepuščeno za *izziv*.

Bloomov filter – s štetjem

Podatkovna struktura:

- imamo k d-bitnih polj `BFi[0..m-1]`, kjer je $m = \frac{cn}{k}$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`:

```
Insert(S, x):
  za vsak i= 1..k:
    naračunaj li= hi(x);
    BF[li]++
```

Bloomov filter – s štetjem

Podatkovna struktura:

- imamo k d-bitnih polj `BFi[0..m-1]`, kjer je $m = \frac{cn}{k}$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`:

```
Insert(S, x):
  za vsak i= 1..k:
    naračunaj li= hi(x);
    BF[li]++
```

iskanje: `Find(`S`, x)`: *izziv*: kaj vrniti?

Bloomov filter – s štetjem

Podatkovna struktura:

- imamo k d-bitnih polj `BFi[0..m-1]`, kjer je $m = \frac{cn}{k}$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`:

```
Insert(S, x):
  za vsak i= 1..k:
    naračunaj li= hi(x);
    BF[li]++
```

iskanje: `Find(`S`, x)`: *izziv*: kaj vrniti?

Časovna zahtevnost: $O(k)$, prostorska zahtevnost: $O(dm) = O(n)$ bitov.

Bloomov filter – s štetjem

Podatkovna struktura:

- imamo k d-bitnih polj `BFi[0..m-1]`, kjer je $m = \frac{cn}{k}$;
- imamo k različnih razpršilnih funkcij $h_1(), h_2(), ..., h_k()$, ki slikajo v domeno $0..m-1$;

in potem operaciji: Podatkovna struktura:

dodajanje: `Insert(`S`, x)`:

```
Insert(S, x):
  za vsak i= 1..k:
    naračunaj li= hi(x);
    BF[li]++
```

iskanje: `Find(`S`, x)`: *izziv*: kaj vrniti?

Časovna zahtevnost: $O(k)$, prostorska zahtevnost: $O(dm) = O(n)$ bitov.
Izkaže se, da je $d = \log \log n$ z zelo veliko verjetnosto dovolj.

Operacije nad BF

Recimo, da imamo slovarja, ki sta predstavljena z Bloomovima filtroma S_1 in S_2 in naj bodo vse $h_{1,i}() = h_{2,i}()$.

- unija neštevnih slovarjev $S_1 \cup S_2$:

```
Union(S1, S2):
  za vsak i= 0..m-1:
    result.BF[i]= S1.BF[i] or S2.BF[i]
```

Operacije nad BF

Recimo, da imamo slovarja, ki sta predstavljena z Bloomovima filtroma S_1 in S_2 in naj bodo vse $h_{1,i}() = h_{2,i}()$.

- unija neštevnih slovarjev $S_1 \cup S_2$:

```
Union(S1, S2):
  za vsak i= 0..m-1:
    result.BF[i]= S1.BF[i] or S2.BF[i]
```

- unija števnih slovarjev $S_1 \cup S_2$: *izziv*

Operacije nad BF

Recimo, da imamo slovarja, ki sta predstavljena z Bloomovima filtroma S_1 in S_2 in naj bodo vse $h_{1,i}() = h_{2,i}()$.

- unija neštevnih slovarjev $S_1 \cup S_2$:

```
Union(S1, S2):
  za vsak i= 0..m-1:
    result.BF[i]= S1.BF[i] or S2.BF[i]
```

- unija števnih slovarjev $S_1 \cup S_2$: *izziv*
- krčenje velikosti neštevnega slovarja z m na $\frac{m}{2}$ bitov: naredimo unijo zgornje in spodnje polovice bitne tabele; ohranimo vse $h_i()$ a odslej ne uporabljamo najbolj pomembnega bita

Operacije nad BF

Recimo, da imamo slovarja, ki sta predstavljena z Bloomovima filtroma S_1 in S_2 in naj bodo vse $h_{1,i}() = h_{2,i}()$.

- unija neštevnih slovarjev $S_1 \cup S_2$:

```
Union(S1, S2):
  za vsak i= 0..m-1:
    result.BF[i]= S1.BF[i] or S2.BF[i]
```

- unija števnih slovarjev $S_1 \cup S_2$: *izziv*
- krčenje velikosti neštevnega slovarja z m na $\frac{m}{2}$ bitov: naredimo unijo zgornje in spodnje polovice bitne tabele; ohranimo vse $h_i()$ a odslej ne uporabljamo najbolj pomembnega bita
- krčenje velikosti števnega slovarja z m na $\frac{m}{2}$ bitov: *izziv*

Izziv: zakaj je zgornje res?

Primeri uporabe

- slovar za angleški črkovalnik
- porazdeljene baze podatkov: izmenjava samo BF in ne celotnih zapisov
- P2P prekrivna omrežja
- meritve tokov podatkov (paketi ali sporočila v omrežjih, borzni podatki ipd.)

Zahtevnost

	prostor	`Find`	`Insert`	`Delete`
seznam	$n + rn$	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$n + rn$	$O(n)$	$O(n)$	$O(n)$
binarno drevo	$n + 2rn$	$O(n)$	$O(n)$	$O(n)$
AVL drevo	$n + 2rn$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
B drevo	$n + brn$	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
RB-drevo	$n + 2rn$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
preskočna vrsta	$n+?rn$	$O(\log n)$	$O(\log n)$	$O(\log n)$
razpršena tabela	$n+?rn$	$O(1)$	$O(1)$	$O(1)$
Bloomov filter	$\frac{cn}{\lg n}$	$O(1)$	$O(1)$	???