Algoritmi in podatkovne strukture – 2

Slovar

Razpršene tabele

Razpršena tabela

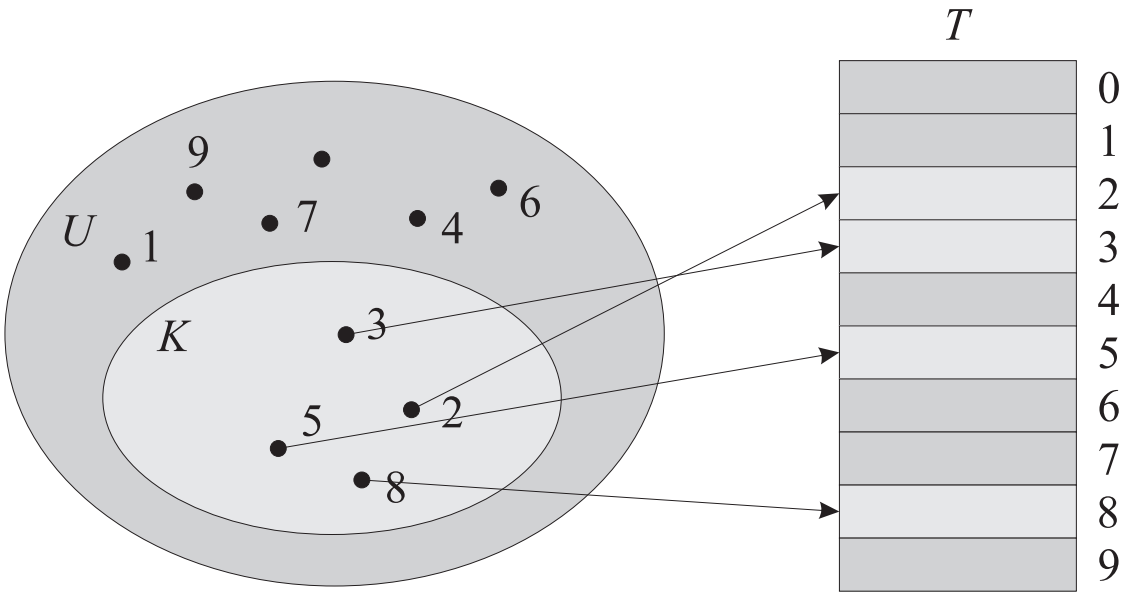
Lahko jo tudi imenujemo zgoščena tabela.

- Ključi, s katerimi imamo opravko 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

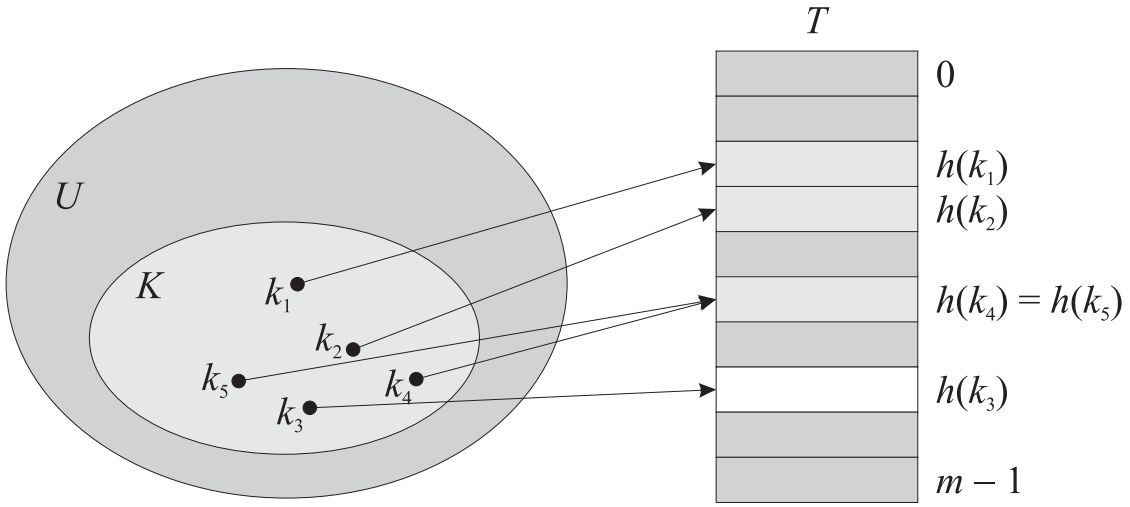
$$h : U \to \{0, 1, 2, \ldots, m-1\}$$

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 $m = O(n)$.

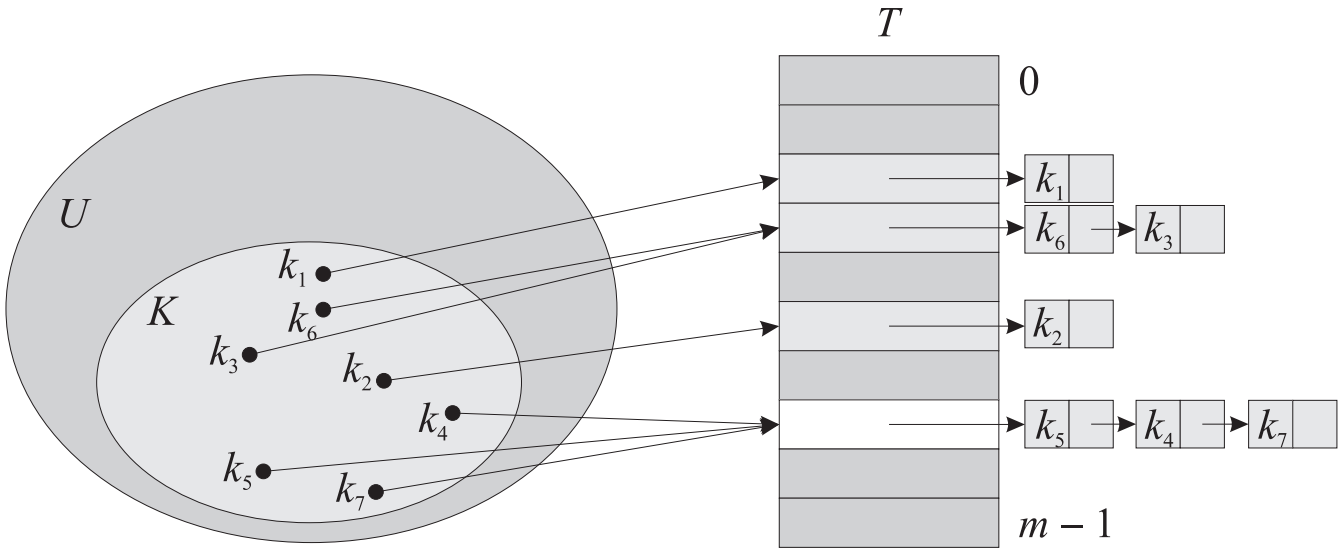
Reševanje sovpadanja

- 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 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[i]= 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 $h' = h(x, 0)$ in $h_i = h(x, i)$, 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 $(i + 1)/n$, 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: $(h'(k) + \mathbf{ih"(k)}) \bmod m$ 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 16, 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?

Zapletenost

	`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 $O(n)$.