

Abstraktni podatkovni tipi in podatkovne strukture

Vrsta s prednostjo, Zgoščena tabela, Bloomov filter, Disjunktne množice

Tomaž Dobravec, Algoritmi in podatkovne strukture 2

Vrsta s prednostjo

Vrsta s prednostjo

✧ Urejena množica elementov

- elemente lahko med seboj primerjamo po velikosti;
- dovolj je, da imamo eno operacijo, recimo `bigger()`

✧ Urejeno množico elementov S shranimo v slovarju, v katerem imamo poleg običajnih operacij (`insert`, `find`, `delete`) tudi naslednje operacije:

- `min(S)` ... vrne najmanjši element množice S
- `deleteMin(S)` ... iz podatkovne strukture odstrani najmanjši element

Z operacijama dostopamo do elementa z najvišjo prioriteto.

✧ Podatkovni strukturi s temi operacijami pravimo “vrsta s prednostjo”

✧ **Želja:** hitre operacije `find()`, `insert()`, `delete()` in `deleteMin()` ter ekstra-hitra operacija `min()`.

Strukturi lahko dodamo še operacije

- `decreaseKey(S, x, d)` ... elementu x zmanjša vrednost ključa za d
- `left(S, x)` ... vrne največji element, ki je še manjši od x
- `right(S, x)` ... vrne najmanjši element, ki je večji od x

Implementacije vrste s prednostjo

- ✧ Najpreprostejša in najpogostejša implementacija: dvojiška kopica
 - uporabimo min-kopico (oziroma, za “obratno” vrsto s prednostjo, max-kopico)
 - operacija $\text{min}()$ vrne koren kopice $\rightarrow O(1)$
 - ostale operacije v kopici potrebujejo $O(\log n)$ časa.
- ✧ Naprednejša (in posledično zahtevnejša) implementacija: Fibonaccijeva kopica
 - uporablja gozd dreves in zahtevne operacije za vzdrževanje strukture,;
 - amortizirana časovna zahtevnost operacij je boljša v praksi z dvojiško kopico;

	<i>min</i>	<i>deleteMin</i>	<i>insert</i>	<i>decreaseKey</i>
Dvojiška kopica	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$
Fibonaccijeva kopica	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$

- ✧ Če imamo v zaporedju v vstavljanj (*insert*) in spreminjanja ključev (*decreaseKey*) in b brisanj (*delete*), bo (amortizirana) časovna zahtevnost pri uporabi dvojiške kopice enaka $O((v+b) \log n)$, pri uporabi Fibonaccijeve kopice pa $O(v + b \log n)$.
- ✧ Vendar: zaradi zapletene strukture je Fibonaccijeva kopica kljub boljši teoretični časovni zahtevnosti v praksi včasih celo slabša od dvojiške kopice.

Zgoščena tabela

Zgoščena tabela

- ✧ **Problem:** imamo elemente (ključ, vrednost), ki bi jih radi shranili v podatkovno strukturo s kar se da hitrim vstavljanjem/brisanjem/iskanjem. Do sedaj nam je uspelo vse operacije narediti v logaritemskem času. Gre še hitreje?
- ✧ Predpostavke:
 - imamo n elementov (k_i, v_i) za $i=1\dots n$,
 - ključi so elementi množice $U=\{0,1,\dots,u-1\}$.
- ✧ Ideja: za shranjevanje uporabimo tabelo T velikosti $|U|$, v kateri ima vsak element določeno mesto.
- ✧ Časovno je taka struktura idealna!
- ✧ Težava: ker je v praksi $n \ll |U|$, je struktura prostorsko ZELO potratna.

Zgoščena tabela

- ✧ Rešitev težave: uporabimo tabelo velikosti $m \ll |U|$, mesto na tabeli, kamor shranjujemo element s ključem k , pa določimo s funkcijo $h()$:

$$h: U \rightarrow \{0, 1, \dots, m-1\}, \quad h: k \rightarrow h(k) \quad \dots \text{zgoščevalna funkcija}$$

Def.: Podatkovno strukturo, ki uporablja tabelo velikosti m ($m \ll |U|$) in zgoščevalno funkcijo za računanje indeksov, imenujemo **zgoščena tabela**.

Implementacija?

Zgoščena tabela - sovpadanja

Ker je število mest v tabeli manjše od števila ključev ($m \ll |U|$), se lahko zgodi, da se dva ključa preslikata v isti indeks.



sovpadanju (collision)

✧ Število sovpadanj je odvisno od distribucije vhodnih podatkov in od zgoščevalne funkcije

✧ Sovpadanju se ne moremo povsem izogniti, zato

A) iščemo funkcijo, pri kateri bo v povprečju čim manj sovpadanj:

- razpršilna funkcija;
- metoda deljenja;
- metoda množenja.

B) iščemo načine, s katerimi se rešujemo, ko do sovpadanja pride.

- veriženje;
- prenaslavljanje.

Zgoščena tabela – zgoščevalna funkcija

- ✧ Od dobre zgoščevalne funkcije pričakujemo, da bo ključe enakomerno **razpršila** po tabeli.
- ✧ Z drugimi besedami: če imamo nek ključ k in poljubni dve mesti i in j v tabeli, potem si želimo, da je enako verjetno, da se bo k preslikal v i ali v j .

Definicija: naj bo $P(k)$ verjetnost, da se bo med ključi pojavil ključ k .

Funkcija $h: U \rightarrow \{0, \dots, m-1\}$, za katero za vsak $j=0, 1, \dots, m-1$ velja

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m}$$

se imenuje **razpršilna funkcija**.

V nadaljevanju bomo uporabljali le zgoščevalne funkcije, ki so razpršilne.

Primeri zgoščevalnih funkcij

- ✧ **Metoda deljenja:** $h(k) = k \bmod m$
 - zelo hiter izračun
 - število m je treba premišljeno izbrati
 - težava: kaj če so vsi ključi potence števila m ?

- ✧ **Metoda množenja:** $h(k) = k * p \bmod m$ (kjer je p vnaprej izbrana konstanta)
 - izbira števila m tu ni kritična
 - za p je dobro izbrati (veliko) praštevilo

Ali obstaja idealna razpršilna funkcija ?

Reševanje problema sovpadanja - veriženje

- ✧ Elemente, ki se preslikajo v isto celico tabele (v tem primeru jo imenujemo koš; bucket) shranjujemo v povezanem seznamu.
- ✧ Vse operacije so sestavljene iz dveh delov: najprej poiščemo pravi koš ($\text{koš} = h(\text{ključ})$), nato v seznamu, ki je pripet temu košu iščemo/dodajamo/brišemo.
- ✧ Časovna zahtevnost?

Reševanje problema sovpadanja - prenaslavljanje

- ✧ Težava veriženja: poleg tabele potrebuješ dodatno podatkovno strukturo (seznam).
- ✧ Kaj pa, če želimo vse elemente shraniti v tabeli?
Pomagamo si s prenaslavljanjem (open addressing): računam zaporedje indeksov v tabeli, dokler ne najdem takega, ki še ni zaseden.
- ✧ Želje:
 - do prostega indeksa bi rad prišel v čim manjšem številu korakov,
 - indeksi se računajo tako, da bom najkasneje v m korakih pregledal celotno tabelo.
- ✧ Težave:
 - prostor v tabeli je omejen (tabele ne morem širiti, ko je polna);
 - brisanje elementov

Za potrebe prenaslavljanja definiramo dvo-parametrično zgoščevalno funkcijo:

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

Linearno prenaslavljanje

✧ Imamo "osnovno funkcijo zgoščevanja" $h'()$:

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

in definiramo dvo-parametrično zgoščevalno funkcijo

$$h(k, i) = \begin{cases} h'(k) & i = 0 \\ (h(k, i-1) + 1) \bmod m & \text{sicer} \end{cases}$$

Kvadratno prenaslavljanje

✧ Imamo "osnovno funkcijo zgoščevanja" $h'()$:

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

in dve neničelni konstanti c_1 in c_2 . Definiramo dvo-parametrično zgoščevalno funkcijo

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$$

Dvojno prenaslavljanje

✧ Imamo dve različni osnovni funkciji zgoščevanja, $h'()$ in $h''()$:

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

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

in definiramo dvo-parametrično zgoščevalno funkcijo

$$h(k, i) = (h'(k) + ih''(k)) \bmod m$$

Število poskusov pri prenaslavljanju

Število poskusov prenaslavljanjem preden najdemo pravo mesto v tabeli je odvisno:

- ✧ od lastnosti zgoščevalne funkcije in
- ✧ od zasedenosti tabele.

Znamo povedati, koliko bomo "v povprečju" iskali, preden bomo našli prosto mesto?

Časovna zahtevnost

	find()	insert()	delete()
dinamična tabela	$O(n)$	$O(n)$	$O(n)$
urejena tabela	$O(\log n)$	$O(n)$	$O(n)$
seznam	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$O(n)$	$O(n)$	$O(n)$
preskočni seznam	$O(\lg(n))^*$ <small>*pričakovani čas</small>	$O(\lg(n))^*$ <small>*pričakovani čas</small>	$O(\lg(n))^*$ <small>*pričakovani čas</small>
dvojiško iskalno drevo	$O(h)=O(n)$	$O(h)=O(n)$	$O(h)=O(n)$
AVL drevo	$O(h)=O(\lg n)$	$O(h)=O(\lg n)$	$O(h)=O(\lg n)$
B-drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
Rdeče-črno drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
Zgoščena tabela	$O(1)$	$O(1)$	$O(1)$

Zgoščena tabela v Javi

- ✧ Vsak objekt v javi ima svoj `hashCode()`, ki predstavlja ključ tega objekta
 - `hashCode()` je definiran na nivoju razreda `Object`
 - `hashCode()` od `Integer` -> vrednost,
 - `hashCode()` od `String` -> string zapisan v 31-tiškem sistemu

Java definira dve zgoščeni tabeli: `HashTable` in `HashMap`

Hashtable

- ✧ sinhronizirana;
- ✧ sovpadanje rešuje z veriženjem; koš je implementiran kot kazalčni seznam;
- ✧ v enem košu so vsi elementi, ki imajo isti `hashCode()`;
- ✧ index (koš) računa po metodi deljenja ($\text{index} = \text{hashCode()} \% \text{array.length}$)
- ✧ začetna velikost tabele: 11, povečevanje (rehash): $\text{size} = 2 * \text{size} + 1$

HashMap

- ✧ nesinhronizirano
- ✧ sovpadanje rešuje z veriženjem; koš je implementiran kot drevo
- ✧ index (koš) računa po metodi deljenja: $\text{index} = \text{hashCode()} \& \text{array.length}$

Bloomov filter

Bloomov filter

- ✧ Bloomov filter je podatkovna struktura, v katero dodajamo podatke, struktura pa nam zna odgovoriti na vprašanje, ali smo nek podatek dodali ali ne.
 - Odgovor NE (tega podatke ni v strukturi) je povsem zanesljiv, pravilnost odgovora DA (ta podatek je v strukturi) pa je obtežena z neko verjetnostjo.

Bloomov filter podpira operaciji

`insert(k)` ... v slovar vstavi element s ključem `k`

`find(k) → true/false` ... v slovarju poišči ključ `k` in vrni `true`, če je `k` v slovarju in `false`, če ni; dovolimo, da je odgovor `true` včasih napačen (*false positive*).

Implementacija Bloomovega filtra

- ✧ Podatke hranimo v tabeli bitov (bitno polje) velikosti m
 - v praksi: če je n predvideno število podatkov, ki jih bomo shranili v filtru, potem je $m=c*n$ (pri čemer je c majhno (5, 6, 7 ...) naravno število);
 - prisotnost elementa v tabeli označimo tako, da prižgemo k karakterističnih bitov v tabeli BF

 - prisotnost elementa preverim tako, da preverim njegove karakteristične bite

- ✧ za računanje karakterističnih bitov potrebujemo k zgoščevalnih funkcij $h_1(), h_2(), \dots, h_k()$, ki slikajo v domeno $0, \dots, m-1$.

Kako vstavimo in kako preverimo prisotnost elementa v strukturi?

✧ **Katere zgoščevalne funkcije lahko uporabim?**

- zahteva: funkcije morajo biti neodvisne in morajo **enakomerno** razpršiti
- Primerne funkcije:
 - kriptografske zgoščevalne funkcije
 - murmur

✧ Potrebujem k različnih funkcij. Kje naj jih dobim?

- lahko vzameš katerekoli od zgoraj naštetih
- vse zgornje uporabljajo seme (seed); če uporabiš dve različni semeni, z isto funkcijo dobiš neodvisne rezultate; primer: murmur(seed1) in murmur(seed2) sta neodvisni funkciji
- funkcije lahko kombiniraš (Kirsch-Mitzenmacher-Optimization).

✧ **Kakšna je verjetnost napačnega DA odgovora?**

Verjetnost napačnega odgovora DA

A grid of horizontal blue lines for data entry, with a vertical red line on the left side.

Brisanje v Bloomovem filtru in primeri uporabe

- ✧ **Kaj se zgodi, če želimo nek podatek iz Bloomovega pobrisati?**

- ✧ Če “ugasnemo” karakteristične bite, dobimo dodaten ”false negative” efekt – morda smo te bite ugasnili še kateremu drugemu elementu!

- ✧ Možne rešitve:
 - namesto tabele bitov BF uporabljamo števec; ko dodam element, povečam števec, ko element brišem, števec zmanjšam;

 - dodaten filter, v katerem shranjujem elemente, ki smo jih zbrisali.

- ✧ **Primeri uporabe Bloomovega filtra**
 - Iskanje podatka v podatkovni bazi.
 - Detekcija škodljivih URL naslovov
 - Antivirusni program.
 - ...

Bloomov filter s štetjem

- ✧ Podobno kot osnovni Bloomov filter, le da nas v tem primeru zanima, kolikokrat je bil nek element vstavljen v strukturo.
- ✧ Izberemo prag (treshold) ϕ in se sprašujemo: ali je bil element v strukturo vstavljen vsaj ϕ -krat. Odgovor, ki ga lahko dobimo je: Zagotovo NE ali Verjetno DA

Implementacija Bloomovega filtra s štetjem?

- ✧ Primer uporabe:
 - zasedenost uporabniškega imena;
 - porabljenost žetonov v igralnici.

Disjunktne množice

Disjunktne množice

- ✧ Imamo podatke, ki so razporejeni v skupine (vsak podatek je v **eni od** skupin).
- ✧ **Iščemo podatkovno strukturo, ki bo shranila oboje: podatke in informacijo o tem, v katero skupino posamezen podatek sodi.**

- ✧ Podatke bomo shranili v **množici množic** elementov

$$S = \{S_1, S_2, \dots, S_n\}$$

ki so paroma disjunktne (t.j. vsak element nastopa **samo v eni** množici).

- ✧ Vsaka množica bo imela svojega **predstavnika**.

Predstavnik množice in operacije disjunktne množice

- ✧ Če nas zanima, ali sta dva element v isti množici, primerjamo predstavnika njunih množic → če se ujemata, sta, sicer nista v isti množici

- ✧ **Kdo je lahko predstavnik množice?**
 - načeloma je to lahko katerikoli element;
 - v nekaterih aplikacijah izbiramo najmanjši/največji element za predstavnika;
 - pomembno: dokler se struktura ne spremeni, mora predstavnik ostati isti;
 - ko se struktura spremeni (ko, na primer, združimo dve disjunktne množici), se predstavnik množice, ki ji pripada nek element, lahko spremeni.

- ✧ Podatkovna struktura "disjunktne množice" ima tri operacije:
 - `makeSet(x)`
 - `find(x)`
 - `union(x, y)`

Primer uporabe operacij disjunktne množice

Izvedba disjunktih množic s kazalcem na predstavnika

- ✧ **Osnovna zahteva:** za vsak element $s \in \cup_i S_i$ potrebujemo podatek o tem, kdo je predstavnik množice, v kateri se s nahaja
- ✧ **Možna izvedba:** vsak element dobi kazalec na predstavnika množice
 - vsi elementi "kažejo" na predstavnika
 - predstavnik "kaže" sam nase
- ✧ Časovna zahtevnost take implementacije?
 - `makeSet(x)`
 - `find(x)`
 - `union(x, y)`

Izvedba z gozdom disjunktih množic

✧ Posamezno disjunktno množico implementiramo kot drevo, v katerem otroci kažejo na očeta:

- obrnjeno drevo: namesto da oče kaže na otroke, otroci kažejo na očeta;
- vsak element x dobi podatek $\text{parent}(x)$;
- oče ne ve, kdo so njegovi otroci; otrok ve, kdo je njegov oče;
- **v korenu tega drevesa je predstavnik množice.**

✧ Časovna zahtevnost take implementacije?

- `makeSet(x)`
- `union(x, y)`
- `find(x)`

✧ **Kako popraviti težavo počasne operacije `find(x)`?**

Kako popraviti težavo počasne operacije `find(x)`?

Izvedba strukture z gozdom disjunktних množic