.Vaje APS2 / 2022 / 05
Teden: 14. – 18. Mar. 2022

Naloga 1

Izvedite amortizirano analizo operacije *insert()* za dinamično tabelo po metodi:

a) vsote,
b) kopičenja, in
c) potenciala.

Naloga 2

V tabeli tabel podrobneje obravnavamo predvsem operaciji *insert()* in *find()*, operacija ***delete()*** pa tipično ni podana. Ugotovili smo tudi že, da si lahko pri operaciji *find()* pomagamo z min/max informacijo, ki je naravno vsebovana v vsaki urejeni pod-tabeli.

Pri tej nalogi razmislite in podajte smiselno in učinkovito implementacijo operacije ***delete()***, ki ohranja »dobre« lastnosti tabele tabel glede na operaciji *insert()* in *find()* v smislu časovne zahtevnosti.

(Obnovimo tudi razmislek in definicijo operacij *insert()* ter *find()* s pomočjo min / max informacije.)

Izhodišče:

Tabela tabel je podatkovna struktura, ki vsebuje k pod-tabel A_i ($i = 0 .. k-1$) fiksnih velikosti 2^i. Osnovna implementacija predpostavlja, da so pod-tabele bodisi prazne ali popolnoma polne, kar pomeni, da si pri tabeli tabel z n elementi lahko pomagamo z binarno predstavitvijo števila n za predočitev informacije, katera pod-tabela je prazna (binarna števka 0) ali polna (binarna števka 1). Pod-tabele so vedno urejene, med njimi pa ni posebnih relacij, na primer:

Pod-tabela	Velikost pod-tabele	Elementi
A_0	2^0	.
A_1	2^1	2 17
A_2	2^2	. . . .
A_3	2^3	1 4 11 11 17 22 23 37
.	.	.
.	.	.
.	.	.
A_{k-1}	2^{k-1}	3 5 7 11 12 28 29 33 ... 92

Ustrezna binarna predstavitev zasedenosti pod-tabel v gornji tabeli tabel z n elementi bi bila:

1 .. 1010,

kjer LSB (najmanj pomemben bit) predstavlja pod-tabelo A_0, MSB (najbolj pomemben bit) pa A_{k-1}.

Operacija *insert()*, klasični postopek:

Element, ki ga vstavljamo naj tvori začetno začasno tabelo $A_{tmp} = [e]$.

Nato obiskujemo pod-tabele od $i = 0$ do $k-1$:

> če je pod-tabela A_i prazna, ji dodelimo A_{tmp} in, če je $i > 0$, še "izpraznimo" vse pod-tabele od A_{i-1} do A_0 , ter **uspešno zaključimo** vstavljanje …
>
> sicer v A_{tmp} urejeno zlijemo A_{tmp} in A_i, ter nadaljujemo na A_{i+1} …

V kolikor so vse pod-tabele neprazne, potem tvorimo novo, dodatno pod-tabelo A_k (velikosti 2^k), ji dodelimo A_{tmp} in, ker je $i > 0$, še "izpraznimo" vse dosedanje pod-tabele od A_{k-1} do A_0, ter **uspešno zaključimo** vstavljanje.

Časovna zahtevnost operacije *insert()* je ***O(n)***, ker v najslabšem primeru z vstavljenim elementom *[e]* urejeno zlijemo vse A_0 do A_{k-1} pod-tabele v novo, dodatno pod-tabelo A_k.

Operacija *find()*, klasični postopek:

Z iskanim elementom *e* obiskujemo pod-tabele od $i = 0$ do $k-1$:

> če je pod-tabela A_i prazna, nadaljujemo na A_{i+1} …
>
> če je *e* enak **min** ali **max** elementu pod-tabele A_i, potem **uspešno zaključimo** iskanje …
>
> če je *e* manjši od **min** ali večji od **max** elementa pod-tabele A_i, nadaljujemo na A_{i+1} …
>
> v pod-tabeli A_i izvedemo dvojiško iskanje elementa *e* in, če ga najdemo, **uspešno zaključimo** iskanje, sicer nadaljujemo na A_{i+1} …

V kolikor smo do sem pregledali vse pod-tabele, potem lahko iskanje **neuspešno zaključimo**.

Časovna zahtevnost operacije *find()* je ***O(lg²n)***, ker v najslabšem primeru element *e* dvojiško iščemo v vsaki od A_0 do A_{k-1} pod-tabel.

Operacija *delete()*, razmislek in končni predlog:

Nekajkrat smo se že spomnili, da je operacija brisanja ali odstranitve elementa v splošnem pravzaprav iskanje elementa z »neugodnim« izzidom za ta iskani element, v kolikor se le-ta v podatkovni strukturi sploh nahaja.

To pomeni, da pri operaciji *delete()* postopamo najprej enako kot pri operaciji *find()*, nato pa je tipično potrebno – če smo element dejansko našli in ga »odstranili« iz podatkovne strukture – še obnoviti značilnosti strukture, ki se lahko »pokvarijo« ravno zaradi izbrisa najdenega elementa.

Pri tabeli tabel smo opazili, da je vstavljanje elementa enakovredno binarnemu povečanju števila *n* (elementov) za 1. Na primer: ob vstavitvi enega elementa v tabelo tabel z 11 elementi, jih imamo na

koncu 12, kar z vidika zasedenosti pod-tabel pomeni 1011 (11) + 1 = 1100 (12) . Če bi na operacijo brisanja elementa gledali kot na binarno odštevanje za 1, bi to pomenilo, da moramo ob izbrisu elementa iz neke pod-tabele A_i le-to izprazniti, nato pa ustrezno urediti načeloma vse nad njo (vključno od A_{i-1} do A_0). Res: ob brisanju enega elementa iz tabele tabel oziroma pod-tabele A3 pri 14 elementih, jih bo v tabeli tabel potem le še 13, kar je enakovredno 1110 (14) – 1 = 1101 (13).

Časovna zahtevnost takšnega pristopa bi bila *O(n)*. Pri tem pa velja opomniti, da si z brisanjem nekako »rušimo« praznost ali razpoložljivost začetka tabele tabel za naknadna vstavljanja, kjer smo videli, da je za operacijo *insert()* kljub časovni zahtevnosti O(n) njena amortizirana cena odličnih O(1)!

Smo lahko nekako boljši?

Če se odločimo za »lažno« – ali bolje rečeno – »leno« brisanje (**lazy delete**), kjer najdenega elementa dejansko ne izvržemo iz podatkovne strukture ampak ga le označimo kot brisanega, potem smo s časovno zahtevnostjo pri takšni operaciji delete na istem kot pri iskanju, torej $O(lg^2 n)$.

Hkrati pa se (ne)zasedenost pod-tabel ne spremeni in še naprej velja amortizirana cena za *insert()*.

Ampak, pozor! Ker se nam sedaj v tabeli tabel v pod-tabelah na različnih nivojih in mestih lahko pojavljajo »leno« brisani elementi bomo zagotovo primorani ponovno preučiti in ustrezno nadgraditi operaciji *insert()* in *find()*. Pa poglejmo...

Pri operaciji *find()* lahko postopamo enostavno enako kot do sedaj, le ob morebitni najdbi iskanega elementa bomo previdni in preverili, če ni slučajno že »leno« izbrisan.

Operacijo *insert()* pa bi veljalo, ob kratkem razmisleku glede učinkovitosti, spremeniti tako, da se pred dejanskim vstavljanjem elementa *e* najprej vprašamo kakšno je stanje pod-tabel A_0 do A_{k-1} glede na dejansko zasedenost.

Natančneje, če gremo z indeksom od 0 proti k-1 in:

- po uvodnem zaporedju dejansko zasedenih pod-tabel najprej naletimo na prazno pod-tabelo A_i, potem se splača izvesti klasično vstavljanje z zlivanjem in urejanjem pod-tabel A_0 do A_{i-1}, sicer pa

- bomo nekje v tem zaporedju pod-tabel, po dejansko zasedenih, najprej naleteli na pod-tabelo A_i z delno zasedenostjo (takšno torej, ki vsebuje vsaj en »leno« brisan element, hkrati pa tudi ni popolnoma prazna), kar pomeni, da se splača urejeno vstaviti v to pod-tabelo le element *e* (lahko bi rekli tudi urejeno zliti ali kaj podobnega, še za malenkost bolj učinkovitega).

Spomnimo se, da je nekaj dodatnega razmisleka potrebno še do opredelitve kako upoštevamo ali ne upoštevamo morebitnih večkratnih pojavitev elementa *e* (po domače: duplikati).

Na odločitev, kako z duplikati znotraj podatkovne strukture upravljamo, so namreč neposredno povezane podrobnosti dejanske implementacije.

S tem so vse tri osnovne operacije nad podatkovno strukturo tabele tabel definirane. Za zaključek si še grafično predočimo razmerja *O(n)*, $O(lg^2 n)$, in *O(lg n)*, pa... Veselo na delo!

Grafična primerjava funkcij *y(n) = n* (modra), .. $lg^2 n$ (rjava) in .. *lg n* (zelena):