Adatstruktúrák, Fibonacci-kupac

Hajnal Péter

Bolyai Intézet, TTIK, SZTE, Szeged

2020. ősz

Adatstruktúrák

Adatstruktúrák adatok struktúrált tárolására szolgálnak.

Minden adatnak van egy azonosítója/címe és egy numerikus (legtöbbször pozitív valós) kulcsa.

A tárolás mellett az adatstruktúra szolgáltatásokat ad az adatok kezelőinek.

Ilyen szolgáltások lehetnek `Beszúr`$(a, \mathcal{S})$, `Töröl`$(a, \mathcal{S})$, `MinimumTöröl`$(\mathcal{S})$, `KulcsCsökkent`$(a, \mathcal{S}, \delta)$.

Szolgáltatások

$\mathcal{S}$ az adatstruktúra, a egy adat.

`KulcsCsökkent`$(a, \mathcal{S}, \delta)$ szolgáltatás

A `KulcsCsökkent`$(a, \mathcal{S}, \delta)$ szolgáltatás kérésének teljesítése azt jelenti, hogy $\mathcal{S}$-et módosítjuk úgy, hogy az a adat kulcsa δ-val csökken (a csökken szó a $\delta > 0$ feltételt „rejti"). A többi adat közben változatlanul megmarad a struktúrában.

`MinimumTöröl`$(\mathcal{S})$ szolgáltatás

A `MinimumTöröl`$(\mathcal{S})$ szolgáltatás kérésének teljesítése azt jelenti, hogy $\mathcal{S}$-et módosítjuk úgy, hogy a minimális kulcsú adat törlődik a struktúrából. A többi adat közben változatlanul megmarad a struktúrában.

Ezekhez a változtatásokhoz/szolgáltatások teljesítéséhez „szervezésre" van szükség.

Fa-kupac

A fa-kupac struktúrában az adatok egy gyökeres fa csúcsaiban helyezkednek el úgy, hogy

(K) Minden adat kulcsa legalább annyi mint gyerekeiben tárolt adatok kulcsai.

Ezt a (K) tulajdonságot kupac tulajdonságnak nevezzük. Ennek következménye, hogy minden csúcs leszámazottjaiban tárolt adatok kulcsai sem lehetnek nagyobbak mint saját kulcsa.

Kupac a memóriában

Egy $\mathcal{F}$ fa-kupac egy pointer, ami a gyökér adathoz mutat.

Tehát a tárolt számok (általánosabban az adat teljes könyvelését tartalmazó rekord) és a fa csúcsai párba vannak állítva.

A fa egy csúcsa és a benne tárolt rekord x_i kulcsa egymás szinonimája.

Ha a fa bináris/ternáris, estleg lefoka korlátozott, akkor minden csúcs/adat/rekord-ban lehet egy mutató gyerekeihez (ha lefoka legfeljebb t, akkor az első, második, ..., t-edik gyerekre mutató pointer).

Ha azonban a fa lefokai nem korlátozottak, akkor az adatot tartalmazó rekord — amit szeretnénk konstans méretűnek tartani — már nem képes az összes gyerek pointerét tárolni.

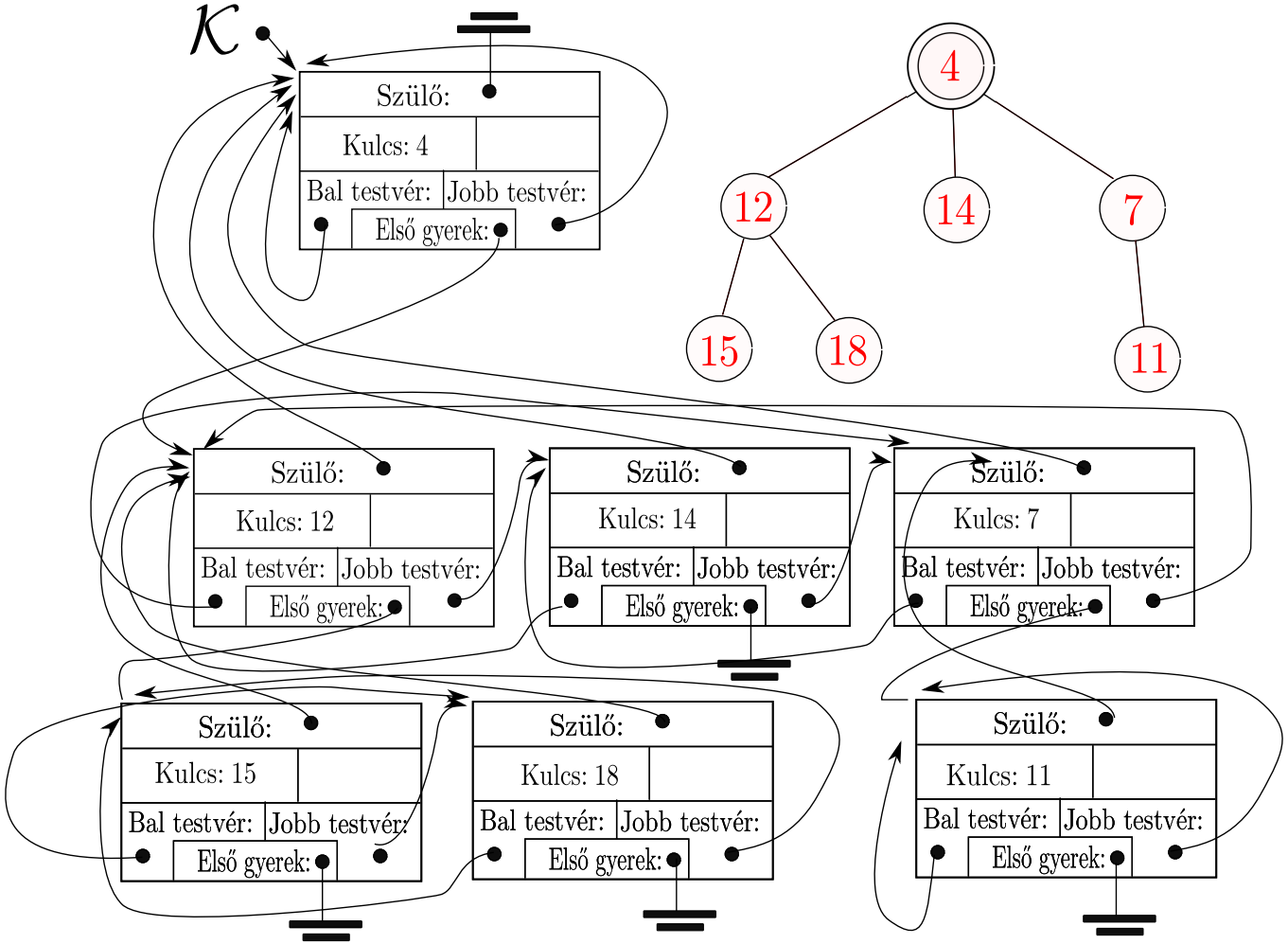
Kupac a memóriában (folytatás)

Így ekkor csak egyetlen gyerekre („elsőszülött") mutat pointer, viszont az összes gyerek (akik testvérek) egy duplán, körszerűen láncolt listában tárolhatók. Így egy szülő az elsőszülött gyerekéhez menve, a testvér listát követve érheti el összes gyerekét.

Tehát minden rekord egy `szülő`, `első szülött`, `következő testvér`, `előző testvér` pointert tartalmaz, hogy a fastruktúra nyilvánvaló legyen.

És ott van még a `kulcs` numerikus értéke és a későbbiekben tárgyalandó további információk. Tehát a rekord belső struktúrája, csak a tárgyalás legvégére alakul ki teljességében.

Kupac a memóriában: Kép



Egy $\mathcal{K}$ kupac szerkezete mint adatstruktúra és mint egy kombinatorikus objektum (piros számok)

A kupac minimumának meghatározása

Egy p pointernél $@p$-vel jelöljuk azt az adatot, amire p mutat.

$@p[kulcs]$ a p által mutatott adat kulcsa, egy numerikus érték.

Legyen $\mathcal{K}$ egy kupac. Ekkor a kupac tulajdonság miatt $@\mathcal{K}[kulcs]$ a kupacban tárolt legkisebb kulcs.

Azaz a tárolt értékek minimima $\mathcal{O}(1)$ költséggel „kinyerhető” a kupacból.

A Fibonacci-kupac

A Fibonacci-kupacnak egy egyszerűsített változatát ismertetjük. Ebben három szolgáltatást támogatunk: `Beszúr`, `MinTöröl` és `KulcsCsökkent`.

Az adatstruktúra fa-kupacok duplán láncolt listája.

Az egyes fa-kupacokban van információnk az ott tárolt kulcsokról: minden csúcsban a kulcs legfeljebb akkora mint a leszármazottakban.

A különböző fákban tárolt adatok azonban „függetlenek”. Megköveteljük a következő tulajdonságot:

(F) Maga a Fibonacci-fa egy pointer a minimális kulcsot tartalmazó fa-kupac gyökeréhez.

Azaz egy Fibonacci-kupacban tárolt legkisebb kulcs nagyon egyszerűen magtalálható: a kupac neve egy adatra mutat, ez egy kupac gyökerének „címe”.

Az alapkérdés

Üres kupaccal indulunk és a szolgáltatások egy sorozatát hajtjuk végre.

n a `Beszúr` szolgáltatások száma lesz.

Azaz n minden pillanatban felülről becsli a tárolt adatok számát.

Szervezzük meg az adatok tárolását és a szolgáltatások kielégítését, hogy a teljes futás „gyors” legyen.

A rang

Definíció

Egy tetszőleges pillanatban vegyük egy éppen a struktúrában szereplő adatot. Ennek rangja a gyerekei aktuális száma.

Definíció

A Fibonacci-kupacot alkotó egyik kupac rangja a gyökerének rangja.

A rang egy dinamikusan változó érték lesz.

A cél

Célunk lesz, hogy bármelyik r rangú fa-kupac által tárolt adatok száma legalább α^r legyen alkalmas $\alpha > 1$ konstansra.

Így seciálisan a fa-kupacok rangja az egész algoritmus során $\mathcal{O}(\log n)$ lesz.

L

egyen R az adatstrúktúránk adatainak legnagyobb rangja az egész futás során.

Tehát bármelyik pillanatban bármelyik fa-kupacból bármelyik csúcsot nézzük, annak rangja legfeljebb R lesz.

Ha a kijelölt célt teljesítjük, akkor $R = \mathcal{O}(\log n)$ teljesül.

Szünet

Beszúr megvalósítása

Ez egy egyszerű lépés lesz. Az új adatot úgy tekintjük mint egy egy csúcsú kupac.

Ezt a kupacok duplán láncolt listájába felvesszük.

A $\mathcal{F}$ Fibonacci-kupac egy gyökérre mutat. Emellé tesszük az új kupacot (amely egyetlen egy csúcst/adatot tartalmaz: az új adatot, amelyet éppen beszúrunk).

Egy kis óvatosság kell. $\mathcal{F}$-nek a legkisebb kulcsú adatra kell mutatni.

Szerencsére a korábbi kulcsok minimuma adott: $@\mathcal{F}[kulcs]$.

Így $@\mathcal{F}[kulcs]$ és $@a[kulcs]$ összehasonlítása meghatározza hová mutasson a bővített Fibonacci-kupac neve. $@\mathcal{F}$ mellé való beszúrás és $\mathcal{F}$ esetleges átállítása $\mathcal{O}(1)$ művelet:

Beszúr megvalósítása (folytatás)

$p := @F[JobbGyerek]$
$@\mathcal{F}[JobbGyerek] := a$
$@a[JobbGyerek] := p$
$@a[BalGyerek] := \mathcal{F}$
$@p[BalGyerek] := a$
If $@a[Kulcs] < @\mathcal{F}[Kulcs]$ then $\mathcal{F} := a$

Azaz az $\mathcal{O}(1)$ becslés igazából hat soros programot jelent.

Ez legfeljebb hét darab elemi utasítás számát takarja: az első öt utasítás elemi lépés, ha a hatodik sor összehasonlítása megfelelően jön (a hatodik elemi lépés), akkor egy további (hetedik elemi lépés) értékadás zárja le az implementációt.

MinTöröl opráció: Naív fázis

$\mathcal{F}$ megmutatja a törlendő elem helyét.

Ennek ElsőGyereke pointere az összes gyereket tartalmazó duplán láncolt lista kezdete. Ezek a gyerekek mindegyike egy fa-kupac gyökere.

Ezen fa-kupacokat (duplán láncolt listában) és az $\mathcal{F}$ többi fa-kupacait (nem a minimális kulcsú gyökérrel rendelkezőket, amelyek egy csonkított duplán láncolt listában vannak miután a minimális kulcsú elemet töröltük) egy listává fűzzük össze.

Ennek költsége $\mathcal{O}(1)$.

Hátra van az új Fibonacci-kupac kupac-listájából a minimális elem kiválasztása: Végighaladunk a listán és az addig látott, gyökérben tárolt kulcsot (helyével együtt) megjegyezve a teljes végigpásztázás után az új Fibonacci-kupac $\mathcal{F}$ címét „kiszámoljuk” (naív megvalósítás).

MinTöröl operáció: Billentési fázis

Billentési fázis

A fa-kupacaink között megszabadulunk az azonos rangúaktól. Azaz a `MinTöröl` szolgáltatás után igaz lesz, hogy fa-kupacaink különböző rangúak.

A kupacok gyökereinek listáját végigpásztázzuk. Ameddig a pásztázásunk elhaladt egy rendbe rakott részt hagyunk magunk után. Itt minden kupac rangja különböző.

Egy $(\rho[i])_{i=0}^{R}$ tömböt nyitunk, amely minden eleme egy pointer lesz. $\rho[r]$ pointerek a fa-kupacaink azon rendszeréről adnak információt, ahol a „rendcsinálás" már megtörtént.

- Ha $\rho[r]$ értéke `nil`, akkor az azt jelenti, hogy nincs r rangú fa-kupacunk.
- Ha $\rho[r]$ értéke nem `nil`, akkor ez rámutat az egyetlen r rangú fa-kupacunkra.

Billentési fázis (folytatás)

A rendcsinálás kezdetén az átfésült rész üres, ρ-ban mindegyik elem a sehová se mutató `nil` pointer lesz.

Ekkor elkezdjük fa-kupacaink végigpásztázását. Az új, aktuális $\mathcal{F}$ fa-kupacnál megnézzük a rangját.

Érdemes az adat rekordjába beleolvasztani a rang paramétert. (Ez kihat a `Beszúr` szolgáltatásra: $@a[rang] := 0$ utasítás is rész lesz.)

Ha a rang r, akkor megnézzük $\rho[r]$-et.

(i) Ha ez `nil`, akkor az új fa-kupacot a rendberakott részbe tesszük, $\rho[r]$ értéke az új fa-kupac lesz, tovább pásztázunk.

(ii) Ha ez NEM `nil`, akkor ez a pointer a rendezett rész egyetlen r rangú $\mathcal{F}'$ fa-kupacára mutat. Ezt kivesszük a rendezett részből és a $\mathcal{F}$, $\mathcal{F}'$ fa-kupacokat billentjük.

Billentés: A definíció

Definíció: Billentés

A *billentés* operáció két fa-kupacból egyetlen fa-kupacot alkot:

A két fa-kupac közül az adja az új gyökeret, amelybe a gyökérbeli/legkisebb tárolt szám a kisebb.

Ennek leszármazottjai megmaradnak a korábbi szerkezetben.

Új gyerekként hozzávesszük a másik fa-kupac gyökerét, amelynek szintén megmarad a leszármazott struktúrája.
Természtesen az új gyökér rangját 1-gyel megnöveljük.

Billentési fázis

Lehet, hogy a rendbe rakott részben volt $r + 1$ rangú kupac is.

Akkor egy újabb billentést végzünk.

A rendrakás közben a billentéssel kapott kupac rangja nő.

R választása olyan volt, hogy ezt a rang nem haladhatja meg.

Valamikor leállunk és rátérünk a következő rendezetlen kupacra.

A rendezetlen rész csökken, elérünk egy rendezett Fibonacci-kupacot.

Észrevétel

Korábban a törlés utáni minimális elem megtalálásást megvalósítottuk naív módon. Aggódtunk, de nem kellett. A rendcsinálás során a végigpásztázást megtesszük. A minimális elem megtalálása és helyénekmegjegyzése a rendcsinálás folyamatába beolvasztható.

Billentési fázis költsége

Észrevétel

Egy billentés és az ehhez szükséges módosítások költsége $\mathcal{O}(1)$.

A teljes költség kifejezése azonban bonyolult.

Ha a `MinTöröl` szolgáltatás kérése előtt $\mathcal{F}$-ben k fa-kupacunk volt, az $@\mathcal{F}$ adat rangja r volt, továbbá a szolgáltatás végrehajtása után k' kupacunk lett, akkor $(k-1)+r-k'$ billentést végeztünk (ami akár 0 is lehet).

De a két lista egyesítése után a kiinduló $(k-1)+r$ kupac mindegyikére el kellett végezni egy vizsgálatot.

Erre a költségre, ami ezekre a kupacokra esik (még a billentést is beleszámolva) mint *billentési költség* hivatkozunk.

KulcsCsökkent: Bevezetés

Az *a* adat kulcsának csökkentése egyetlen aritmetikai művelet.

A probléma, hogy ez része egy kupacnak és a (K) kupac tulajdonság elromolhat.

A kulcs csökkentése után lehet, hogy adatunk rossz helyen lesz szülője alatt.

Megjegyezzük, hogy lehet, hogy a kulcs csökkentése nem rontja el (K)-t. Ha pedig egy gyökérben csökkentünk kulcsot, akkor nem lehetséges a fenti probléma megjelenése. A továbbiakban arra kell gondolnunk, hogy nem gyökér-csúcsban csökkentünk kulcsot és a kupac tulajdonságot helyre kell hoznunk.

KulcsCsökkent: Naív fázis

A naív megoldás, hogy „szülője alól levágjuk” A módosított kulcsú adatot leszármazottjaival egy új fa-kupacnak tekintjük, apja rangja 1-gyel csökken.

Ismét egy rendetlenség jelentkezhet: a fa-kupacok rangjai közt ismétlődés jelenhet meg, de ezzel nem törődünk. Majd a legközelebbi `MinTöröl` szolgáltatás során ez megjavul.

(F) megtartására is figyelnünk kell. (Könnyű.)

A valódi probléma mélyebb. Sok `Beszúr` operáció és egyetlen `MinTöröl` operáció nagy rangú fa-kupacokat hoz létre. Ha ezek után egy nagy rangú fa-kupac gyökerének minden unokája esetén ennek kulcsát csökkentjük, akkor gyökeres csillaghoz jutunk. A célunk (hogy r rangú fa-kupacban legalább r-ben exponenciális sok szám tárolódjon) nem teljesül.

A megoldás ismét fáradságos szervezés.

KulcsCsökkent: ne vágjunk le túl sok gyereket

Minden adatnál vigyázunk, hogy ne vágjuk le sok gyerekét.

Ehhez információra van szükségünk.

A rekordokba egy Boole komponenst rakunk: „Vágtuk-e már le gyerekét?”. Ha igen és nem gyökér csúcsról van szó, akkor értéke ‘!’, különben ‘∅’.

KulcsCsökkent: Kaszkád vágás

Kaszkád vágás

Egy kulcs csökkentésnél megnézzük a szülő megfelelő komponensét, azaz elvégezzük a $@a[\text{csonkított}] =?$ '!' tesztet.

(i) Ha egyenlő, azaz nem az első gyereket vágtuk le róla, akkor őt is levágjuk szülőjéről, akinek szintén megnézzük a megfelelő bitjét. És megfelelően haladunk tovább, amíg a hatás le nem áll (lásd (ii)). Ezt nevezzük *kaszkád-vágásnak*.

(ii) Ha a kaszkád-vágásnál valamikor egy nem gyökér szülő esetében $@a[\text{csonkított}] =$'$\emptyset$' tapasztalunk, akkor csak átírjuk ezt a bitet '!'-re és leállunk. Ugyancsak leállunk, ha egy gyökeret érünk el. Ennek 'csonkított' komponense érdektelen, nem szükséges írás.

KulcsCsökkent (folytatás)

A levágott csúcsok egy-egy kupacként a módosított $\mathcal{F}$-be kerülnek. Egy kis munka szükséges (F) megtartására. A 'csonkított' bit is elveszti értékét. $\emptyset$ értékre állítjuk.

Egy vágás és „lokális környezétében" végzett munka $\mathcal{O}(1)$.

A teljes költség attól függ, hogy hány vágás történt.

Észrevétel

Lehet hogy sok vágás alkotja a kaszkád vágást, de csak legfeljebb egy adat esetében írjuk a csonkított infot '!' értékre, abban a nem-gyökér csúcsban, ahol a *kaszkád „kihal"*.

Analízis: A Lemma

Definíció

Legyen $\ell(r)$ az a maximális szám, hogy amire teljesül, hogy a Fibonacci-kupac kezelése során végig teljesüljön, hogy egy r rangú csúcs leszármazottjaival együtt legalább $\ell(r)$ adatot tartalmaz.

Nyilván $\ell(0) = 1$ és $\ell(1) = 2$ (először 1 rangú fa-kupacot két 0 rangú billentésével kaphattunk, ami 2 csúcsú, a lehetséges minimális méret).

Lemma

Ha $r \geq 3$, akkor

$$\ell(r) \geq \ell(r-2) + \ell(r-3) + \ldots + \ell(0) + 2.$$

Analízis: Lemma bizonyításának kezdete

Vegyünk egy tetszőleges pillanatot az algoritmus során és egy tetszőleges v csúcsot, amelynek r gyereke van ebben a pillanatban: $v_1, v_2, \ldots, v_r$.

v_i sorsa lehetett bonyolult: v alákerülhetett, majd egy kulcs csökkentésnél megszűnt gyerekének lenne, majd egy későbbi minimális kulcs törlésénél egy billentéssel újból v alá került. Az utolsó gyerekké válás pillanatának időbelisége adja meg az indexelést.

Tehát amikor v_i utoljára alákerült v-nek, akkor v_1, v_2, ..., v_{i-1} már gyerek volt, v rangja legalább $i-1$ volt. Mivel gyerekké válást csak billentés okoz, ezért a gyerekké válás pillanatában v_i is legalább $i-1$ rangú volt.

Mivel v_i most is v alatt van, ezért kaszkád-vágás nem vágta le szüleje alól. Azaz legfeljebb egy gyerekét vágtuk le alóla. Azaz ebben a pillanatban is rangja legalább $i-2$.

Analízis: Lemma indukciós bizonyítása

A csúcsok mélysége szerinti indukcióval bizonyítunk (a legmélyebb csúcsokkal kezdünk és a gyökerek felé haladunk).

A legmélyebb csúcsok levelek, az állítás semmitmondó, teljesül.

Az indukciós lépés nyilvánvaló. v_2, v_3, ..., v_r és leszármazottjai $\ell(0) + \ell(1) + \ldots + \ell(r-2)$ hozzájárulást adnak v leszármazottjaihoz. v és v_1 még két csúcsot ad az összeszámolandó csúcsokhoz.

A lemmát beláttuk.

Analízis: Fibonacci-számok

Definíció

Legyen $F_0 = 1, F_1 = 2$, továbbá $F_n = F_{n-1} + F_{n-2}$ amennyiben $n \geq 2$. Ez a sorozata a Fibonacci-sorozat egy változata (egyes tárgyalásban az alap Fibonacci-sorozat egy eltoltja).

Teljes indukcióval egyszerűen látható, hogy $F_n \geq \left(\frac{1+\sqrt{5}}{2}\right)^n$. Hasonlóan egyszerű indukció adja, hogy

$$\ell(r) \geq F_r \geq \left(\frac{1+\sqrt{5}}{2}\right)^r.$$

A fenti egyenlőtlenség mutatja, hogy célunkat elértük, speciálisan tudjuk, hogy $R = \mathcal{O}(\log n)$. Másrészt magyarázatot is ad arra, hogy egy XX. századi adatstruktúra miért kapta nevét egy sok évszázaddal korábbi matematikusról.

Beszúr amortizált analízise

A `Beszúr` operációra az $\mathcal{O}(1)$ tényleges költsége mellett $\mathcal{O}(1)$ „billentési letétet” is terhelünk.

Ezt a megfelelő kupac gyökere mellé rakott pénzösszegnek gondoljuk.

Így minden kupacnak a Fibonacci-kupacban lesz egy „bilentési tartaléka” a gyökerénél. Ez a tulajdonság az egész algoritmus során megmarad.

(Í): Ígéret

Az algoritmus futásának minden pillanatában a Fibonacci-kupacot alkotó kupacok gyökerénél egy billentési költségnyi letét van.

Tétel

A `Beszúr` operáció amortizációs összköltsége így is $\mathcal{O}(1)$.

`MinTöröl` amortizált analízise

Tétel

`MinTöröl` amortizált költsége $\mathcal{O}(R) = \mathcal{O}(\log n)$.

A törölt elem gyerekeinek a gyökerek listájába fűzése ($\mathcal{O}(1)$ költség).

A kiinduló R hosszú tömb inicializálásának költsége $\mathcal{O}(R)$.

A törölt elem legfeljebb R gyerekéhez is egy billentés költségét elhelyezünk. Ennek költsége is $\mathcal{O}(R)$. (Í)!

Ezek után billentések sorozata következik, amit az alúlra került fa gyökerénél elhelyezett letétből fedezzük.

Az új fa gyökere mellett ott marad még a billentési letét. Azaz az (Í) ígéretünket eddig megtartjuk.

`KulcsCsökkent` amortizált analízise

A `KulcsCsökkent` naív fázisának tényleges költsége aritmetikai (kulcsérték csökkentése), vágási, fa-kupacok közé berakás költsége. Ez $\mathcal{O}(1)$ és vágási költségnek nevezzük.

Mellette letéteket is terhelünk a `KulcsCsökkent` végrehajtására: két billentési letétet és egy vágási letétet „fizettetünk”.

Az amortizált költség még így is $\mathcal{O}(1)$.

Egyik billentési letétet a kulcs-csökkentett és emiatt levágott adat mellé rakjuk. Ez fontos, hisz ez Fibonacci-kupacot alkotó kupacok közé kerül gyökérként és (Í)-hez kell ez a letét.

`KulcsCsökkent` amortizált analízise (folytatás)

A második billentési letét és a vágási letét helye az az adat, amely 'csonkított' paraméterét felülírjuk '!' értékre.

Észrevétel

Igaz lesz, hogy minden '!' állapotú adat mellett ott lesz egy törlési és billentési letét.

Nem kell aggódni a naív fázis után esetleg hosszú kaszkád miatt. A vágásuk költsége letétbe van helyezve.

A vágás után a Fibonacci-kupacot alkotó kupacok közé kerül egy billentési letéttel gyökere mellett. Azaz (Í) továbbra is fennáll.

Tétel

`KulcsCsökkent` amortizált költsége $\mathcal{O}(1)$.

A tétel

Tétel

Egy üres Fibonacci-kupacból kiindulva valamilyen sorrendben n darab `Beszúr`, m darab `MinTöröl` és d darab `KulcsCsökkent` operációt hajtunk végre. Ekkor az algoritmusunk lépésszáma

$$\mathcal{O}(n + m \cdot \log n + d).$$

Szünet

Legrövidebb út problémája

A Dijkstra-algoritmus inputja egy (G, ℓ, s) hármas, ahol G egy egyszerű, összefüggő gráf, $s \in V(G)$ egy csúcs és $\ell : E(G) \to \mathbb{R}_{++}$ egy távolságfüggvény G élein.

Nyilvánvalóan ez a távolságfüggvény minden sétához rendel egy hosszt, mint az élsorozatának éleihez rendelt hosszok összege (ha egy élen többszörösen áthalad a séta, akkor az áthaladás multiplicitásaszor járul hozzá az él hossza a séta hosszához).

A feladat minden $v \in V(G)$ estére meghatározni a legrövidebb sv séta/út hosszát (esetleg egy minimális hosszú út kiszámolása is hozzátartozik a feladathoz).

Dijkstra algoritmusa

Az algoritmus során bizonyos S halmazokra ($s \in S$ mindig teljesülni fog) tudni fogjuk az $t \in S$ csúcsokra az st legrövidebb út hosszát, továbbá azt az információt, hogy ez a leggrövidebb séta megvalósítható S-en belül is. Kezdetben $S = \{s\}$.

Az S-be került csúcsok már „nem hatnak” az algoritmus további futására.

Legyen $N(S)$ azon S-en kívüli csúcsok halmaza melyeknek van szomszédja S-ben. Az $n \in N(S)$ csúcsokra tudjuk azon utak minimális hosszát, amelyek s-ből indulnak, S-en belül haladnak, majd legutolsó csúcsként elérik n-et.

Az $N(S)$-beli csúcsok és címkéjük határozza meg a futás további részét.

Dijkstra algoritmusa (folytatás)

Vizsgáljuk az $N(S)$-ben lévő címkék halmazát. Ezek lesznek egy $\mathcal{A}$ adatstruktúrában tárolva.

Kezdetben $\mathcal{A}$ üres és s szomszédait `Beszúr` operációval $\mathcal{A}$-ba rakjuk (az n szomszéd kulcsa $\ell(sn)$).

Most következik az update-lépés, amikor is S-t növeljük $\mathcal{A}$ minimális kulcsú elemével. $\mathcal{A}$ szempontjából egy `MinTöröl` szolgáltatás jön.

Tegyük fel, hogy az n gráfcsúcs adatát töröltük. n S-en kívüli m szomszédainál történik lényeges változás:

(i) Ha m az adatstruktúrában volt, akkor címkéjét/kulcsát átírhatjuk. Az új (vagy nem is új) $\min\{c(m), c(n) + \ell(nm)\}$ értékre. Ez bizonyos esetben egy `KulcsCsökkent` szolgáltatást jelent.

(ii) Ha m nem volt az adatstruktúrában, akkor $c(n) + \ell(nm)$ kulccsal `Beszúr` operációval bekerül $\mathcal{A}$-ba.

Dijkstra algoritmusa (folytatás)

Ezt az update-lépést ismételjük addig, amíg S a teljes csúcshalmaz nem lesz, amikor is meglesz a kiszámolandó információnk.

A teljes futás során $|V| - 1$ darab `Beszúr` (az összes s-től eltérő csúcs látókörbe kerül) és $|V| - 1$ darab `MinTöröl` (az összes s-től eltérő csúcs kikerul az aktuális $N(S)$-ből) szolgáltatást végzünk. Továbbá a `KulcsCsökkent` operációk száma legfeljebb $|E|$.

Dijkstra algoritmusa Fibonacci-kupacokkal

Tétel

A Dijkstra-algoritmus megvalósítható

$$\mathcal{O}(|V| \log |V| + |E|)$$

lépéssel.

- Ha gráfunk nem ritka, azaz $|E| \gg |V|$, akkor a `MinTöröl` szolgáltatások a ritkák.
- Az algoritmusunk ekkor végez el nagy „rendrakási” lépéseket.
- Az összes többi esetben lustán dolgozik: az éppen szükséges átírások mellett potenciált épít a szükségszerű ismételgetett műveletekhez (lásd billentések, kaszkád-vágás).
- Így a leggyakoribb lépések olcsók lesznek.

Vége van!

Köszönöm a figyelmet!