

1. Adatstruktúrák

Adatstruktúrák adatok struktúrált tárolására szolgálnak. Minden adatnak van egy numerikus (legtöbbször pozitív valós) kulcsa. A tárolás mellett az adatstruktúra szolgáltatásokat ad az adatok kezelőinek. Ilyen szolgáltatások lehetnek $\text{Beszúr}(a, \mathcal{S})$, $\text{Töröl}(a, \mathcal{S})$, $\text{MinimumTöröl}(\mathcal{S})$, $\text{KulcsCsökkent}(a, \mathcal{S}, \delta)$. Értelmezésük nyilvánvaló: \mathcal{S} az adatstruktúra, a egy adat. Például $\text{KulcsCsökkent}(a, \mathcal{S}, \delta)$ szolgáltatás kérésének teljesítése azt jelenti, hogy \mathcal{S} -et módosítjuk úgy, hogy az a adat kulcsa δ -val csökken. A többi adat közben változatlanul megmarad a struktúrában. A $\text{MinimumTöröl}(\mathcal{S})$ 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 nyilván sok olyan munkára is szükség van, amely nem közvetlen a szolgáltatás kielégítését szolgálja, hanem az adatstruktúra szerkezetét szervezi a módosított adatsokaság kezelésére.

2. Fa-kupacok

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

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

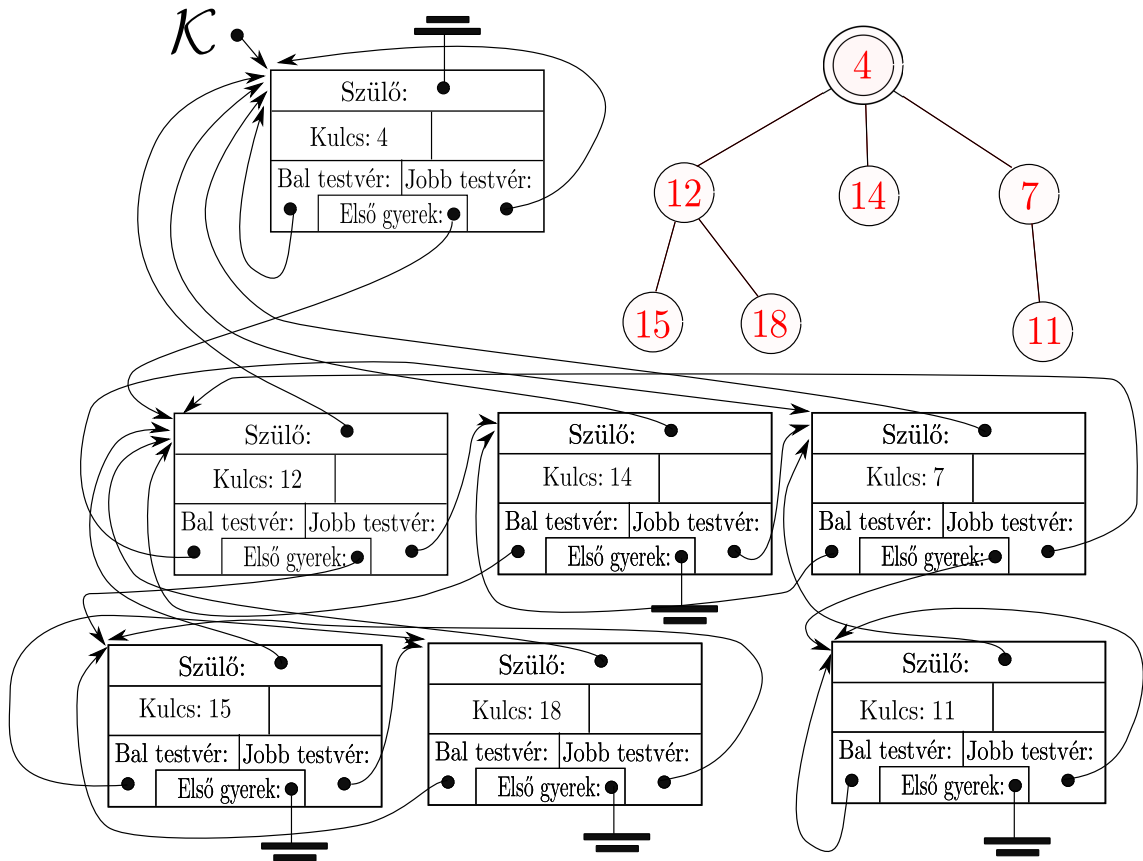
Ezt a (K) tulajdonságot kupac tulajdonságnak nevezzük. Ennek következménye, hogy minden csúcs leszámozottjaiban tárolt adatok kulcsai sem lehetnek nagyobbak mint saját kulcsa.

Egy \mathcal{F} fa-kupac egy pointer, ami a gyöker 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, esetleg lefoka korlátozott, akkor minden csúcs/adat/rekordban lehet egy mutató gyerekeihez (ha lefoka legfeljebb t , akkor az első, második, ..., t -edik gyerekekre 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. Így ekkor csak egyetlen gyereke („elsőszülött”) mutat pointer, viszont az összes gyerek (akik testvérek) egy duplán, körszerűen láncolt listában tárolhatók. Így egy szülő az elsőszülött gyerekehez menve, a testvér listát követve érheti el összes gyerekeit. Tehát minden rekord egy **szülő**, **első szülött**, **következő testvér**, **előző testvér** pointeret 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.



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

Egy p pointernél $@p$ -vel jelöljük 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 minimuma $\mathcal{O}(1)$ költséggel „kinyerhető” a kupacból.

3. 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 fában tárolt adatok azonban „függetlenek”. Maga a Fibonacci-fa egy pointer a minimális kulcsot tartalmazó fa-kupac gyökeréhez. Azaz egy Fibonacci-kupacban tárolt legkisebb kulcs nagyon egyszerűen megtalálható: a kupac neve egy adatra mutat, amiben a tárolt kulcs az összes tárolt szám minimuma.

Egy fa-kupac *rangja* a gyökere gyerekeinek száma. Célunk lesz, hogy az r rangú fa-kupacok által tárolt adatok száma legalább α^r legyen alkalmas $\alpha > 1$ konstansra.

Így speciálisan a fa-kupacok rangja az egész algoritmus során $\mathcal{O}(\log n)$ lesz, ahol n minden pillanatban felülről becsli a tárolt adatok számát. Üres kupaccal indulva n a **Beszúr** szolgáltatások száma lesz. A rangot nem csak a fa-kupachoz rendeljük gyökere alapján. Minden csúcsonak lesz rangja: gyerekeinek száma.

Jelölés. Legyen R az adatstruktúránk fa-kupacainak 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 $R = \mathcal{O}(\log n)$ igaz lesz.

3.1. Beszúr(a, \mathcal{F})

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 gyökerre mutat. Emellé tesszük az új kupacot (amely egyetlen egy csúcsot/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:

```

p := @F[JobbGyerek]
@F[JobbGyerek] := a
@a[JobbGyerek] := p
@a[BalGYerek] := F
If @a[Kulcs] < @F[Kulcs] then F := a

```

Azaz az $\mathcal{O}(1)$ becslés igazából 5 darab elemi utasítás számát takarja.

3.2. MinimumTöröl(\mathcal{F})

\mathcal{F} megmutatja a törlendő elem helyét. Ennek ElsőGyereke pointer az összes gyereket tartalmazó duplán láncolt lista kezdete. Ezek a gyerekek mindegyike egy fa-kupac gyökere. Ezen fa-kupacokat és az \mathcal{F} fa-kupacainak listáját $\mathcal{O}(1)$ lépéssel egy listává fűzhetők össze.

Most azonban egy furcsa dolgot teszünk. A Fibonacci-kupac kezelése egy lassú dolgozó munkája. Eddig nem sokat tettünk a belső struktúra kialakítására. Most azonban egy jelentős átszabást végzünk el: A fa-kupacaink között megszabadulunk az azonos rangúaktól. Azaz a **MinTöröl** szolgáltatás után igaz lesz, hogy fa-kupacaink különböző rangúak.

Ezt a következőképpen tesszük: Egy $(\rho[i])_{i=0}^R$ tömböt nyitunk, amely minden eleme egy pointer lesz. $\rho[r]$ pointerok 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.

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 F fa-kupacnál megnézzük a rangját. Érdekes 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. A *billentés* egy egyszerű operáció, amely a két fa-kupacból egyetlen fa-kupacot alkot: A két fa-kupac közül az adja az új gyökeret, amelybe a 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. Természetesen az új gyökér rangját 1-gyel megnöveljük. A rendberakott rész csökken, még mindig van egy fa-kupacunk, amit a rendezett részbe szeretnénk tenni.

A billentéssel két r rangú fából egy $r + 1$ rangút nyertünk. Most $\rho[r + 1]$ -t olvassuk és az (i)/(ii) közül az érvényes eset szerint lépünk tovább. Előbb-utóbb a rendezett rész bővül és a pásztázással tovább haladunkhatunk. Ha az új fa-kupac lista pásztázása kimerül, akkor megkaptuk a MinTöröl szolgáltatás végrehajtás utáni új fa-kupac listát. Ebben minden fa-kupac rangje különböző lesz.

A fenti leírásban nem törödtünk a Fibonacci-kupac alappointerének updateelésére. Ez nem triviális, de nem jelent kihívást az érdeklődő hallgató számára.

Ha a MinTöröl szolgáltatás kérése előtt k fa-kupacunk volt, a gyökér rangja r volt, továbbá a szolgáltatás végrehajtása után k' kupacunk lett, akkor $(k - 1) + r - k'$ billentést végeztünk. Egy billentés és az ehhez szükséges módosítások költsége $\mathcal{O}(1)$. A billentési költség $\mathcal{O}(k - k' + r - 1)$. A teljes költségben csak a ρ lista inicializálása nincs elszámolva. Ennek költsége $\mathcal{O}(R)$. A teljes költség $\mathcal{O}(R) + \mathcal{O}(k - k' + r - 1)$.

3.3. KulcsCsökkent(a, \mathcal{S}, δ)

Az a adat kulcsának csökkentése egyetlen aritmetikai művelet. A probléma, hogy ez része egy kupacnak és a kulcsa csökkentése elronthatja a kupac szerkezetét (minden adat leszármazottjainak kulcsa legalább akkora mint az adatbeli kulcs). A kulcs csökkentése után lehet, hogy adatunk rossz helyen lesz szülője alatt.

A naív megoldás, hogy „szülője alól levágjuk” (természetesen csak abban az esetben, ha ő nem gyökér, amikor is nem kell semmit sem tennünk): A módosított kulcsú adatot leszármazottjaival egy új fa-kupacnak tekintjük, apja rangja 1-gyel csökken.

Ismét egy rendetlenség jelentkezik: a fa-kupacok rangjai közt ismétlődés jelenhet meg, de ezzel nem törődünk. Majd a legközelebbi MinTöröl szolgáltatás során ez megjavul. 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 unokáinak kulcsait csökkentjük, akkor a célunk (r rangú fa-kupacban legalább r -ben exponenciális sok szám tárolódjon) nem teljesül. Így minden adatnál vigyázunk, hogy ne vágjuk le sok gyereket. A rekordokba egy Boole komponenszt rakunk: „Vágtuk-e már le gyereket?”. Ha igen és nem gyökér csúcsról van szó, akkor értéke '!', különben '.'.

Egy kulcs csökkentésnél megnézzük a szülő megfelelő komponensét, azaz elvégezzük a `@a[csonkított] ? '!'` tesztet.

- (i) Ha egyenlő, azaz nem az első gyereket vágjuk 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 *kaszád-vágásnak*. (A gyökerben ez a információ nem lényeges.)
- (ii) Ha a kaszkád-vágásnál valamikor egy nem gyöker szülő esetében $@a[\text{csonkított}] = \cdot$ tapasztalunk, akkor csak átírjuk ezt a bitet '!'-re és leállunk. Ugyancsak leállunk, ha egy gyökeret érünk el. Ennek csonkított' komponense érdektelen, nem szükséges írás.

A levágott fa-kupacokat a fa-kupacok duplán láncolt listájába rakjuk (ami a Fibonacci-kupac nevének átírását is követelheti, ahogy ezt már korábban láttuk).

Az operáció költsége $\mathcal{O}(f)$, ahol f a vágások száma a kaszkád-vágásban.

3.4. Analízis

Definíció. Legyen $\ell(r)$ az a maximális szám, hogy ami a Fibonacci-kupac kezelése során végig teljesüljön, hogy egy r rangú fa-kupacban legalább $\ell(r)$ adat legyen.

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).

1. Lemma. *Ha $r \geq 3$, akkor*

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

Bizonyítás. Vegyünk egy tetszőleges v csúcsot, amelynek r gyereke van ebben a pillanatban: v_1, v_2, \dots, v_r . Az indexelést úgy választjuk, hogy növekvő sorrendje az v alákerülés időbeliségét kövesse. Egy apróság, de v_i sorsa lehetett bonyolult: v alákerülhetett, majd egy kulcs csökkentésnél megszűnt gyerekének lennie, majd egy későbbi minimális kulcs törlésénél egy billentéssel újból v alá került. Mi az utolsó gyerekké válás pillanatát vesszük, ennek időbelisége adja meg az indexelést. Tehát amikor v_i utoljára alákerült v -nek, akkor v_1, v_2, \dots, v_{i-1} már gyerek volt. Azaz v_i gyerekké válásakor v rangja legalább $i - 1$ volt. Mivel gyerekké válást csak billentés okoz az algoritmusban és billentést csak két azonos rangú fa között végzünk, 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 gyereket vágjuk le alóla. Azaz ebben a pillanatban is rangja legalább $i - 2$.

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

A legmélyebb csúcsok levelek, az állítás semmitmondó, teljesül.

Az indukción lépés nyilvánvaló. v_2, v_3, \dots és leszármazottjai $\ell(0) + \ell(1) + \dots + \ell(r - 2)$ hozzájárulást adnak v leszármazottjaihoz. v és v_1 még két csúcsot ad az összeszámolandó csúcsokhoz. A lemmát beláttuk. ■

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 (egy-egy tárgyalásban az alap Fibonacci-sorozat egy eltolója).

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.

A hátralevő rész amortizált analízis: A **Beszúr** operációra az $\mathcal{O}(1)$ tényleges költsége mellett $\mathcal{O}(1)$ „billentési letétet” is terhelünk. Így minden fának, ami újonnan megjelenik lesz egy „billentési tartaléka”. Az operáció amortizációs összköltsége így is $\mathcal{O}(1)$.

A **MinTöröl** operáció amortizációs költsége $\mathcal{O}(R) = \mathcal{O}(\log n)$, hiszen a billentések költségét a letétből fedezzük. (Az újonnan megjelent adatoknál már láttuk, hogy hozzák billentési tartalékukat. Később látjuk azt is, hogy azok a fák, amiket levágunk és így kerülnek a fa-kupacok listájára azok is hoznak magukkal egy billentési tartalékot.)

A **KulcsCsökkent** tényleges aritmetikai (kulcsérték csökkentése), vágási, fa-kupacok közé berakás költsége mellé terhelünk egy billentési letét (erre szüksége lehet egy esetleges későbbi **MinTöröl** operációnak). A lépésre terheljük a szülő vizsgálatát. Ha itt ‘!’ írására kerül sor, akkor vágási, fa-kupacok közé berakás és egy billentési letétet terhelünk magunkra. Ha a szülőnél nincs ‘!’ írása, akkor vagy leállunk (egy gyökérben vagyunk), vagy már ‘!’ jelet látunk. De akkor ott vannak a letétek, amiből a kaszkad-vágás esedékes költsége fedezhető. Az aktuális terhelés/amortizációs költség $\mathcal{O}(1)$.

2. 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*

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

4. Alkalmazások

A Fibonacci-kupac adatstruktúrájának sok alkalmazása van. Mi csak a Dijkstra-algoritmust vizsgáljuk. Feltesszük, hogy input gráfunk összefüggő.

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

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 legrövidebb séta megvalósítható S -en belül is. 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 eléri n -et.

Az algoritmus kezdetén $S = \{s\}$ és S címkéje 0, ami nyilván helyes: a legrövidebb ss út hossza 0. $N(S)$ tartalmazza s szomszédait. Egy n szomszéd címkéje $\ell(sn)$.

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.

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 **MinTöröl** (az összes s -től eltérő csúcs kikerül az aktuális $N(S)$ -ből) szolgáltatást végzünk. A **KulcsCsökkent** operációk száma legfeljebb $|E|$.

Ha \mathcal{A} -t egy Fibonacci-kupacként valósítjuk meg, akkor kapjuk a következő tételt:

3. Tétel. *A Dijkstra-algoritmus megvalósítható*

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

lépéssel.

A fenti alkalmazás igazából egy motiváció volt a Fibonacci-kupac kitalálásához. 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 ismételtetett műveletekhez (lásd billentések, kaszkád-vágás). Így a leggyakoribb lépések olcsók lesznek.