SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Fibonaccijeva hrpa

Kristijan Verović
Voditelji: *Adrian Satja Kurdija i Marin Šilić*

Zagreb, svibanj 2023.

SADRŽAJ

1. Uvod

Hrpa ili gomila (eng. heap) je poznata struktura podataka koja se najčešće uvodno prikazuje kroz binarnu hrpu (eng. binary heap). U ovom radu će se prezentirati Fibonaccijeva hrpa [6] čije će ime biti objašnjeno u opisu $extractMin$ operacije. Ona ima bolje asimptotske složenosti operacija $insert$, $decreaseKey$ i $union$. Na slici 1.1 nalazi se prikaz složenosti operacija po raznim vrstama hrpa, kod Fibonaccija će operacije $decreaseKey$ i $extractMin$ biti amortizirane, maksimalne složenosti poziva jedne operacije $O(n)$. U ovom radu ćemo ovdje navedene operacija opisati, analizirati i dokazati njihovu složenost.

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[8]	Θ(1)	Θ(log *n*)	*O*(log *n*)	*O*(log *n*)	Θ(*n*)
Leftist	Θ(1)	Θ(log *n*)	Θ(log *n*)	*O*(log *n*)	Θ(log *n*)
Binomial[8][9]	Θ(1)	Θ(log *n*)	Θ(1)[a]	Θ(log *n*)	*O*(log *n*)[b]
Fibonacci[8][2]	Θ(1)	*O*(log *n*)[a]	Θ(1)	Θ(1)[a]	Θ(1)
Pairing[10]	Θ(1)	*O*(log *n*)[a]	Θ(1)	*o*(log *n*)[a][c]	Θ(1)
Brodal[13][d]	Θ(1)	*O*(log *n*)	Θ(1)	Θ(1)	Θ(1)
Rank-pairing[15]	Θ(1)	*O*(log *n*)[a]	Θ(1)	Θ(1)[a]	Θ(1)
Strict Fibonacci[16]	Θ(1)	*O*(log *n*)	Θ(1)	Θ(1)	Θ(1)
2–3 heap[17]	*O*(log *n*)	*O*(log *n*)[a]	*O*(log *n*)[a]	Θ(1)	?

Slika 1.1: Usporedba složenosti operacija različitih struktura hrpa, preuzeto s [4].

Spuštanje složenosti s $O(logn)$ na $O(1)$ barem teoretski pruža poboljšanja u primjenama od čega prvo pada na pamet Dijkstrin algoritam(ili Primov algoritam za minimalno razapinjuće stablo koji je dosta sličan Dijkstri) kojem se složenost pomoću Fibonaccijeve hrpe spušta sa $O((V+E)logV)$ na $O(E+VlogV)$. Može se proučavati i struktura sama za sebe i iz toga izvući potencijalne primjene, recimo slučaj kada je puno $decreaseKey$ operacija i malo $extractMin$ operacija. Primjer takve situacije može se naći u društvenim mrežama gdje se neke objave žele staviti na "trending" te između dva objavljivanja trending sadržaja($deleteMax$) jako puno "lajka-

nja" sadržaja($increaseKey$). Teoretske prednosti se ne precrtavaju uvijek idealno u praktične prednosti, pogotovo kod Fibonaccijeve hrpe koja će imati visoku vremensku konstantu i veliku potrošnju memorije. U sklopu ovog rada implementirat ćemo te potom izmjeriti vremensko i memorijsko ponašanje oba opisana slučaja implementirana u usporedi s korištenjem obične binarne hrpe.

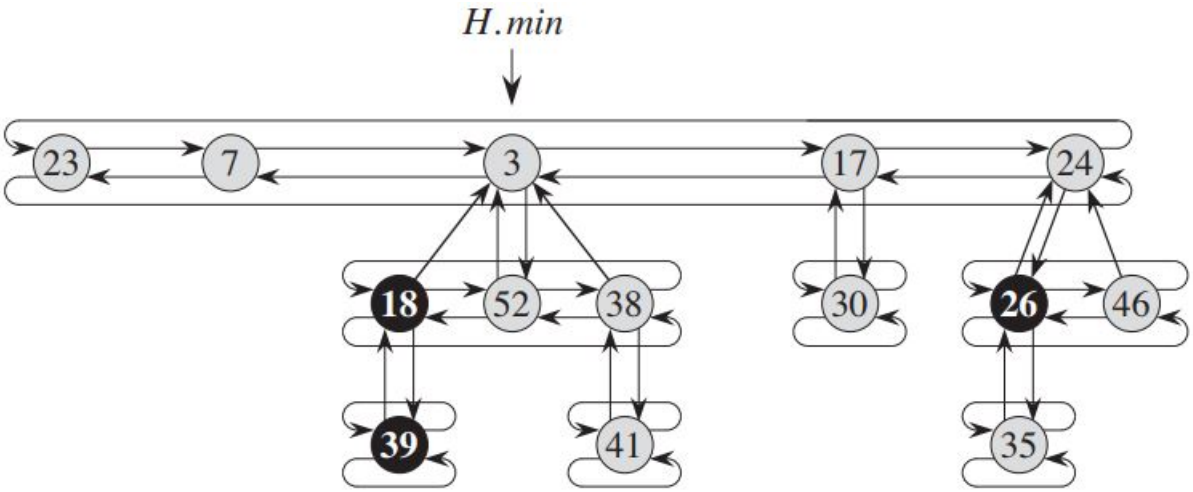
2. Fibonaccijeva hrpa

U ovom poglavlju će se opisati struktura i operacije te u potpunosti ili obrisno dokazati svojstva Fibonaccijeve hrpe. Iako će se opisivati struktura "min-heap", prebacivanje u "max-heap" je lagano jednom kad se shvati način rada. Razlika će biti samo u operatoru usporedbe dvije vrijednosti i u tom da korisnik treba paziti da u $decreaseKey$ korisnik uvijek stavi veće jednaku vrijednost trenutnoj na čvoru što je analogna pretpostavka i za "min-heap" gdje mora staviti manje jednaku vrijednost trenutnoj. Preporuča se uz ovdje opisane elemente pogledati i drugdje dostupne slike ili videozapise u svrhu bolje vizualizacije rada Fibonaccijeve hrpe.

2.1. Struktura

Slika 2.1 prikazuje strukturu neke Fibonaccijeve hrpe, vidimo da se radi o više hrpa spojenih zajedno od kojih jedna sadrži najmanji element[1] te imamo zaseban pokazivač na taj čvor. Zbog toga što imamo direktni pristup samo najmanjem čvoru, operacije $extractMin$, $insert$ i $union$ ćemo izvoditi kraj njega. Sva djeca nekog čvora su povezana dvostruko povezanom listom (eng. doubly linked list), na isti način su povezani i korijenski čvorovi odnosno čvorovi na dubini nula. Neki čvorovi su označeni crnom bojom kao rezultat operacije $decreaseKey$ što će biti pojašnjeno kasnije.

[1] Potencijalno je više čvorova s istom vrijednošću pa i minimalnom, naime hrpa nije matematički skup odnosno može sadržavati više istih vrijednosti.



Slika 2.1: Struktura Fibonaccijeve hrpe in medias res, uzeto iz [5].

Najbolje je prikazanu sliku nadopuniti implementacijskim detaljima prikazanu na kodu 2.1. Svrha varijable info je da znamo kojem djelu neke vanjske strukture pripada cvor jer nam najčešće samo brojčana vrijednost val bez nekog entiteta da ga vežemo uz njega nije korisna. Varijabla marked je rezultat operacije $decreaseKey$ dok su varijable val i deg dovoljno objašnjene komentarima u kodu. Svaki čvor pamti svog roditelja (za čvorove na 0-toj razini nema roditelja pa je nullptr), dok su pokazivači za dvostruko povezanu listu left i right. Čvor pamti samo jedno svoje dijete jer ima sve mogućnosti kao i kad bi pamtio svu djecu, samo što bi to iziskivalo više memorije i operacija spajanja.

Kod 2.1: C++ kod za strukturu cvor

```
template
struct cvor{
    T val; //(iliti key) numericka vrijednost(int, long
       long, double)
    int info;  //oznaka kojem cvoru u npr. Dijkstri
       pripada
    int deg;  //stupanj, broj djece
    bool marked; //oznacen da/ne

    cvor* parent,left,right,child; //pokazivaci

    cvor(T _val, int _info){ //konstruktor
        val = _val;
```

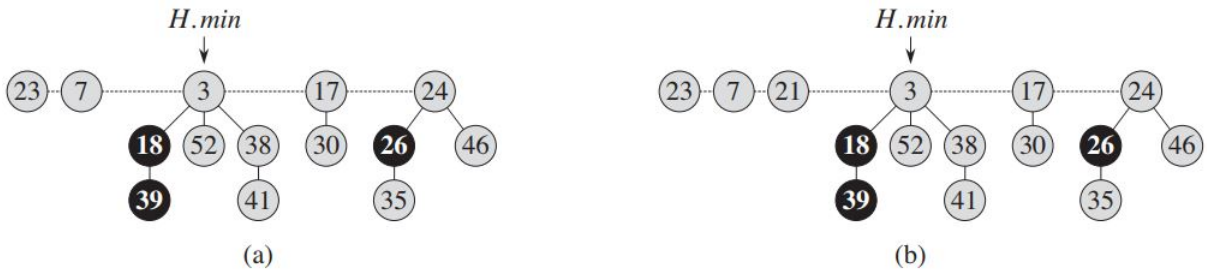
```
        info = _info;
        deg = 0;
        marked = false;
        parent = left = right = child = nullptr;
    }
};
```

Ovdje bi imalo smisla provjeriti koliko memorije troši jedna instanca čvora. Recimo da je T tipa int, i da za bool moramo instacirati 4 umjesto jednog bajta zbog arhitekture računala odnosno riječi (eng. word) u memoriji. Za val,info,deg i marked. to iznosi 4*4=16 bajtova. Pretpostavimo razumno da radimo na 64-bitnom sustavu i da svaki pokazivač zauzima 8 bajtova, budući da ih je 4 to iznosi 4*8=32 bajtova. Dakle jedna instanca strukture cvor iznosi 16+32=48 bajtova. Ekvivalentna obična binarna hrpa bi trebala sadržavati samo varijable val i info odnosno 4+4=8 bajtova, dakle Fibonaccijeva hrpa iziskuje čak 6 puta više memorije. Ovo bi moglo u praksi predstavljati problem usprkos boljih asimptotskih složenosti operacija Fibonaccijeve hrpe, stoga ćemo u poglavlju o performansama to i razmotriti.

2.2. Operacije

2.2.1. Insert

Na slici 2.2 je vizualno prikazano što se događa u operaciji insert. Pozivom operacije insert u korijensku listu se doda novi čvor i potencijalno kaže da je taj čvor novi minimum ukoliko je stvarno najmanji. Budući da ćemo u korijenskoj listi imati samo pokazivač na $minimalniCvor$, spojit ćemo novi čvor odmah do njega te s njegovim susjedom održavajući strukturu dvostruko povezane liste.

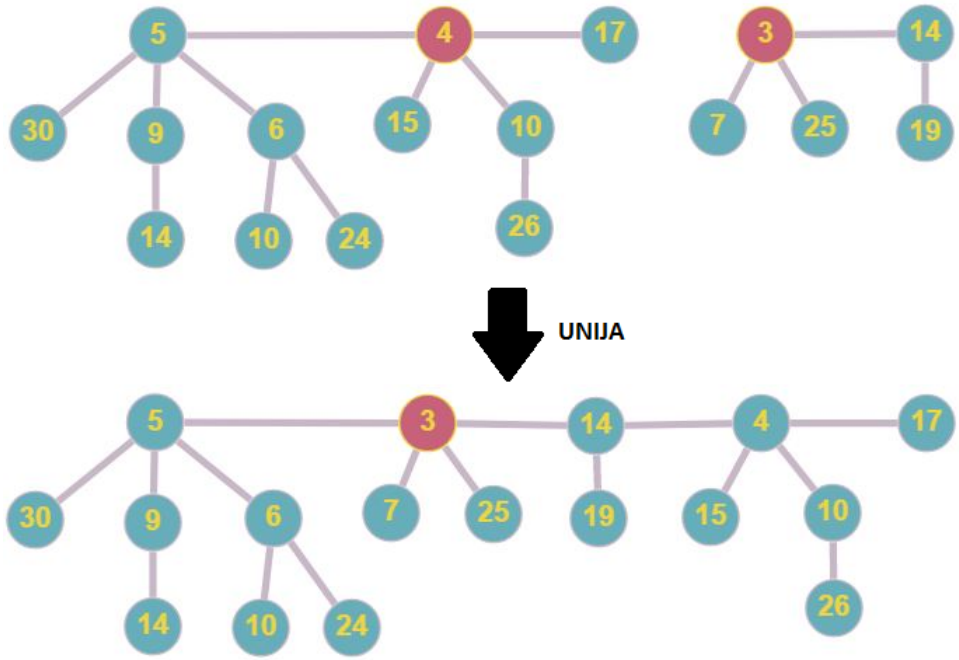


Slika 2.2: Prikaz operacije u kojoj u hrpu a) unosimo čvor s vrijednošću 21 što se vidi na slici b), uzeto iz [5].

Složenost ovog umetanja je naravno O(1), međutim može se dogoditi da nakon puno insert-ova dobijemo dugačak lanac. Operacija $extractMin$ će u svom pozivu amortizirati složenost insert operacija jer će morati taj lanc sažeti na $logn$ čvorova što ćemo vidjeti u nastavku. I dalje je prinos operacije insert ukupnoj složenosti O(1) jer imamo jednu operaciju spajanja+jednu operaciju prestrukturiranja koju na sebe preuzima $extractMin$, a znamo da je O(2) i dalje O(1).

2.2.2. Union

Operacija $union$ je samo spajanje dvaju različitih hrpa zajedno. Ova operacija je praktički ista kao i operacija insert samo što sad potencijalno sudjeluje 4 čvora u prespajanju umjesto 3 kao kod inserta. Prikazana[2] ilustracija 2.3 prikazuje operaciju unije dviju hrpa.



Slika 2.3: Unija dviju hrpa, crvenom bojom označeni minimumi.

Kao i kod inserta, operacija $extractMin$ morat će amortizirati trošak spajanja dviju hrpa u slučaju da opet dobijemo neki lanac, iako ako se dobro razmisli taj se trošak može ponovno izvorno prepisati operaciji insert. Uzevši u obzir da je složenost spajanja dviju hrpa kod osnovne binarne hrpe jednaka $O(n)$, ovo je drastično ubrzanje. Nadalje ovakva jednostavna operacija spajanja prirodno omogućava paralelizam prilikom izgradnje binarne hrpe. Kako ne bismo imali samo lanac, samo na kraju svakog

[2]Crtano u `https://graphonline.ru/en/`

paraleliziranog odsječka napravimo operaciju $extractMin$ i onda ponovno insertamo taj isti minimum. Ista shema se može općenito primijeniti kad god mislimo da imamo predugačak lanac u korijenskoj listi.

2.2.3. DecreaseKey

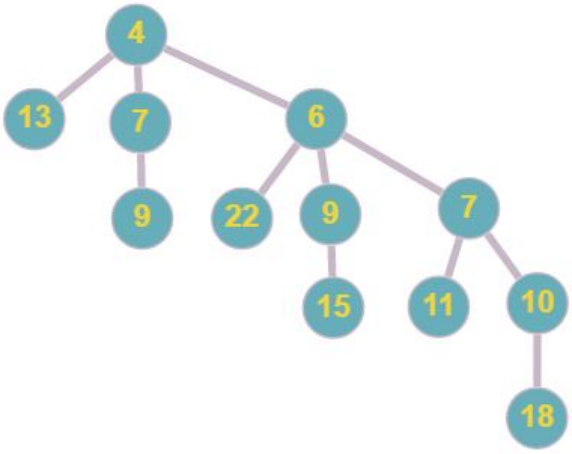
Operacija $decreaseKey$ nešto manje poznata te čak nije implementirana u C++ovom priority_queue iako bi ju binarna hrpa sa složenošću od $O(logn)$ prirodno mogla podržati. $decreaseKey$ smanjuje vrijednost postojećeg čvora na hrpi te nakon toga prestrukturira hrpu po potrebi.

Ovu operaciju najlakše je opisati sljedećim algoritmom:

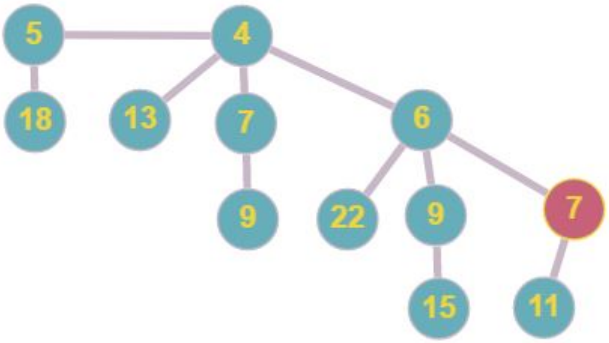
DecreaseKey(čvor,x)

1. Odabrani čvor smanjuje svoju vrijednost na x, ako je x manje od vrijednosti roditelja odreži čvor od roditelja, makni mu oznaku ako uopće postoji i spoji ga zajedno s njegovim podstablima u korijensku listu (u nastavku ovaj slijed operacija zovemo "reži") i idi na 2. korak, inače kraj.
2. Ako čvor nema roditelja ili je roditelj korijenski čvor stani.
3. Sada kaskadno režemo označene roditelje čvora sve dok ne naiđemo na korijenski čvor ili čvor koji je označen kojeg u tom slučaju samo označimo i stanemo.

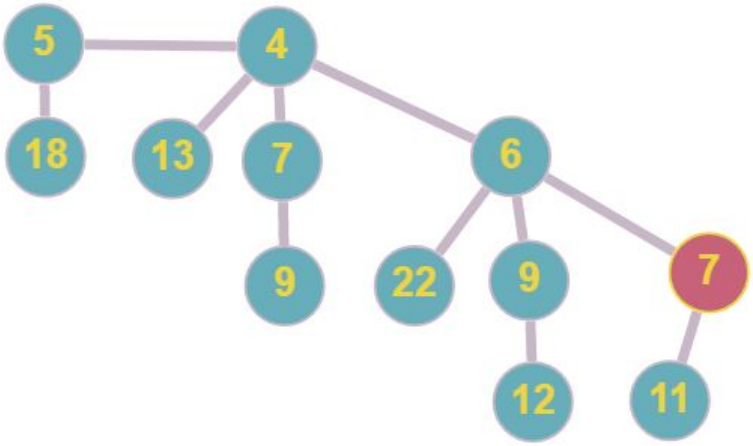
Sljedeće slike počevši od 2.4 vizualiziraju rad $decreaseKey$ operacije, neki ključevi su namjerno isti da se još jednom naglasi mogućnost postojanja više čvorova s istom vrijednošću.



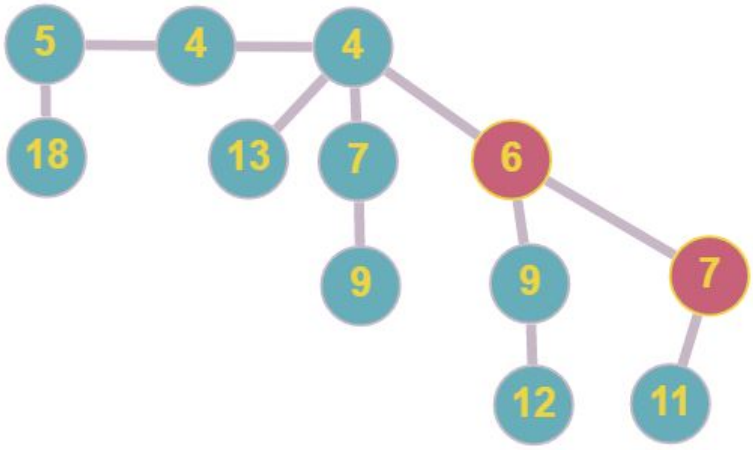
Slika 2.4: Početna Fibonaccijeva hrpa.



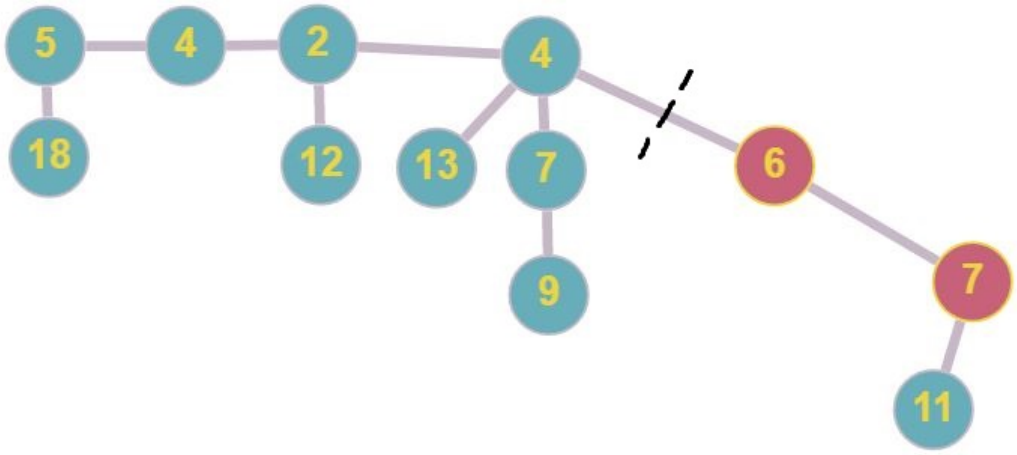
Slika 2.5: Desnom čvoru s vrijednošću 10 vrijednost se smanjuje na 5 što je manje od vrijednosti njegovog roditelja. Zbog toga on označava svog roditelja, biva odrezan i stavljen u korijensku listu. Čvor 7 je prvi put označen i ne propagira dalje označavanje.



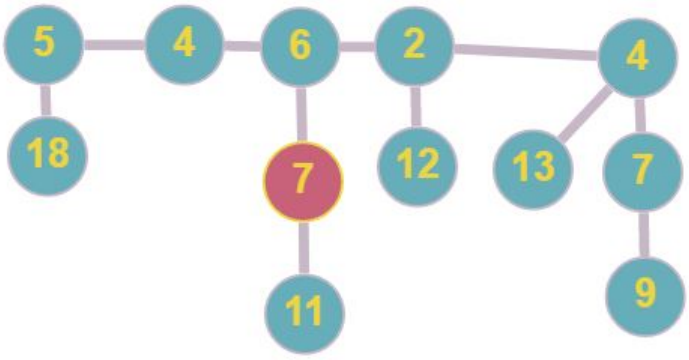
Slika 2.6: Čvor s vrijednošću 15 se smanjuje na vrijednost 12. Budući da je i dalje manji od roditelja ništa se ne događa.



Slika 2.7: Čvor 22 se smanjuje na 4.

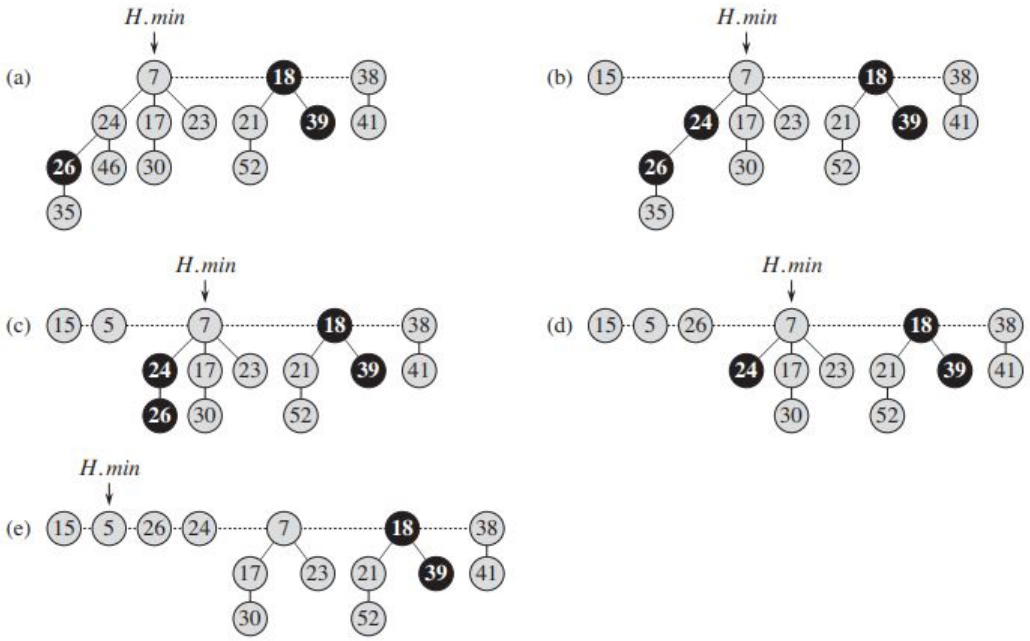


Slika 2.8: Desni čvor 9 se smanjio na 2 i ubacuje u korijensku listu, budući da je njegov roditelj označen, sada i njega treba rezati. Ova slika prikazuje još nedovršenu operaciju prije rezanja čvora 6



Slika 2.9: Dovršetak operacije iz prethodne slike, čvor se odsjeće i odznači te je pojen u korijensku listu.

Vjerojatno je dobro pogledati još jedan primjer operacije $decreaseKey$ koji se nalazi se na slici 2.10.



Slika 2.10: Dva poziva operacije $decreaseKey$. **(a)** Početna Fibonaccijeva hrpa. **(b)** Čvor 46 s vrijednošću 46 se smanjuje na 15 i potom se spaja u korijensku listu te njegov roditelj biva označen. **(c)–(e)** Čvor s vrijednošću 35 se smanjuje na 5. U (c) dijelu čvoru se smanji vrijednost i biva smješten u korijensku listu. U (d) dijelu se reže roditelj početnog čvora te se nastavlja dalje u (e) kaskadno rezanje. Na kraju u (e) se početni čvor stavlja kao minimum (detalj implementacije, moglo se to i odmah na početku reći). Preuzeto iz [5]

Dokaz složenosti

U uvodu se na prikazu 1.1 tvrdi da je amortizirana složenost operacije $decreaseKey$ O(1), idemo vidjeti zašto je to tako.

Dio složenosti operacije $decreaseKey$ amortizira operacija $extractMin$, naime kada smanjimo vrijednost čvora ispod vrijednosti roditelja režemo ga i stavljamo u korijenski čvor te time delegiramo jednu bazičnu operaciju $extractMin$-u, naime povećava broj korijenskih čvorova što povećava složenost te operacije za 1 osnovnu operaciju prespajanja. Označimo li broj ukupnih $decreaseKey$ operacija do nekog trenutka sa E, složenost prouzrokovana od E je $O(E)$ ako ubrajamo samo do sada opisane posljedice $decreaseKey$ operacije. Nastavimo s drugom posljedicom i pogledajmo njen prinos složenosti.

Na početku nema označenih čvorova, izvođenjem $decreaseKey$ operacija neki čvorovi potencijalno budu označeni dok izvođenjem drugih operacija postojeće oznake se ili ne mogu promijeniti ili im se broj smanji. Proučavamo li svaki put kada smo morali proći neki kaskadni lanac označenih čvorova, mi smo samo na sebe pre-

uzeli složenost prošlih $decreaseKey$ operacija i to $O(1)$ za svaku operaciju iz koje je nastala trenutna označenost. S ovime smo i dalje na $O(E)$ i Budući da smo pokrili sve izvore složenosti, ovime je dokaz gotov.

2.2.4. ExtractMin

Algoritam operacije će počivati na činjenici (koju ćemo kasnije pokazati) da svaki čvor može imati asimptotski najviše $O(logn)$ djece, konkretno možemo uzeti $3 + 2 * logn$ kao sigurnu gornju ogradu iako se možda može dokazati da je manja, ali je i dalje logaritamska. Algoritam slijedi:

ExtractMin()

1. Obrišemo minimalni čvor i njegova podstabla unesemo u korijensku listu. Ponovo iteriramo po korijenskoj listi kako bismo našli novi minimum.

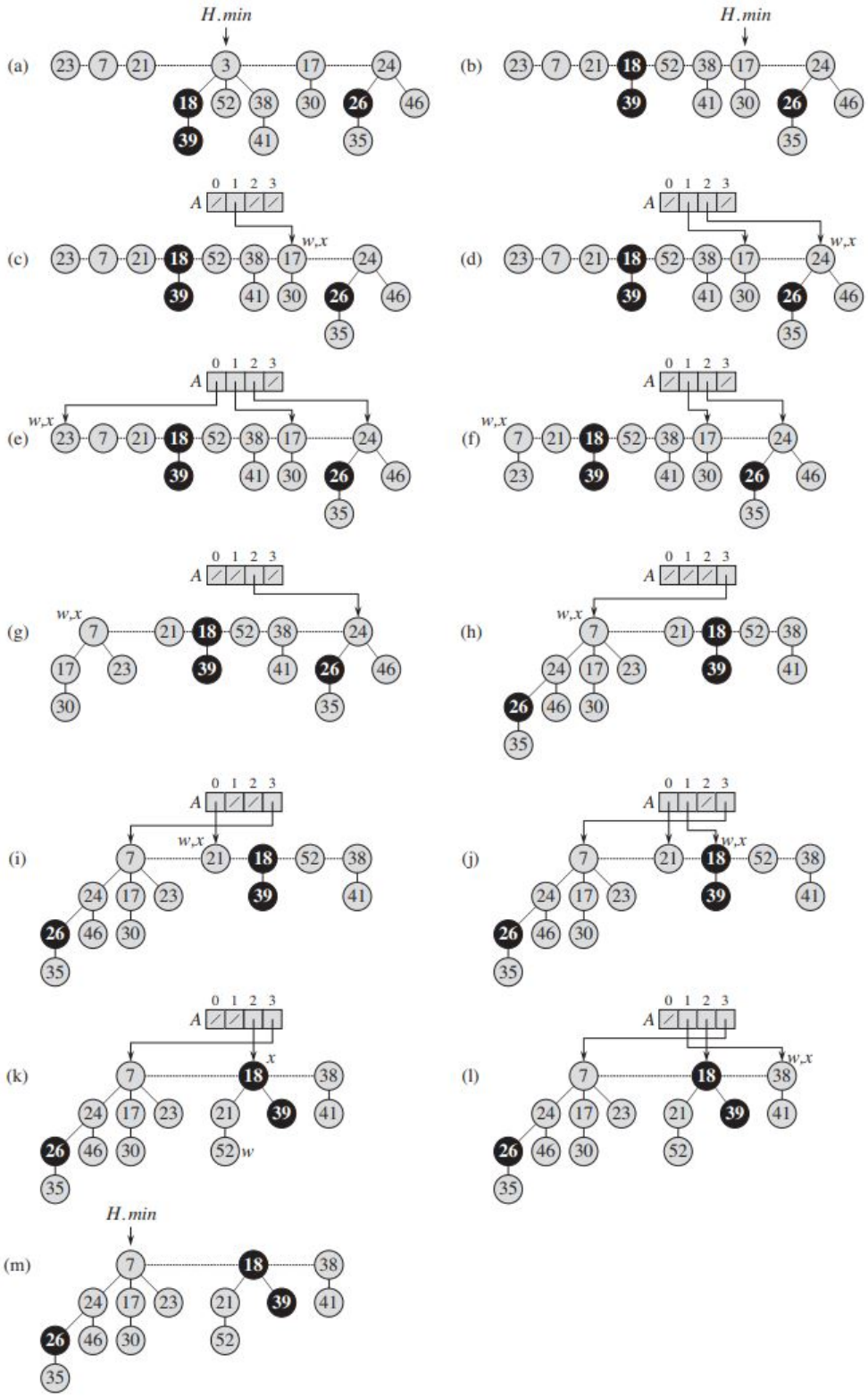
2. Napravimo pomoćni niz pokazivača pom na čvor duljine $3+2*logn$. Postavimo cijeli niz na nullptr vrijednost.

3. Iteriramo od minimalnog čvora i idemo u recimo desno (mogli smo i u lijevo) sve dok opet ne naiđemo na minimalni čvor. Dakle, ako je n čvorova, bit će n iteracija u kojoj u svakoj idućoj pristupamo desnom čvoru. Neka je deg stupanj čvora na kojem se trenutno nalazimo.

4. Ako u pom ne postoji čvor stupnja deg onda stavljamo trenutni čvor u $pom[deg]$ i nazad na korak 3. Inače korak 5.

5. Imamo dva čvora istog stupnja. Sada kažemo da je onaj veće vrijednosti dijete onog manje vrijednosti. Uzimamo manji čvor kao onaj s kojim ćemo raditi dalje, $pom[deg]$ potaje nullptr, deg se povećava za jedan i idemo na korak 4. Valja primjetiti da će se jednoj iteraciji ovo često kaskadno događati.

Vizualni opis operacije je na slici 2.11. Ovdje iteracije nakon brisanja čvora ne počinju iz novog minimalnog čvora, ali je logika ista,



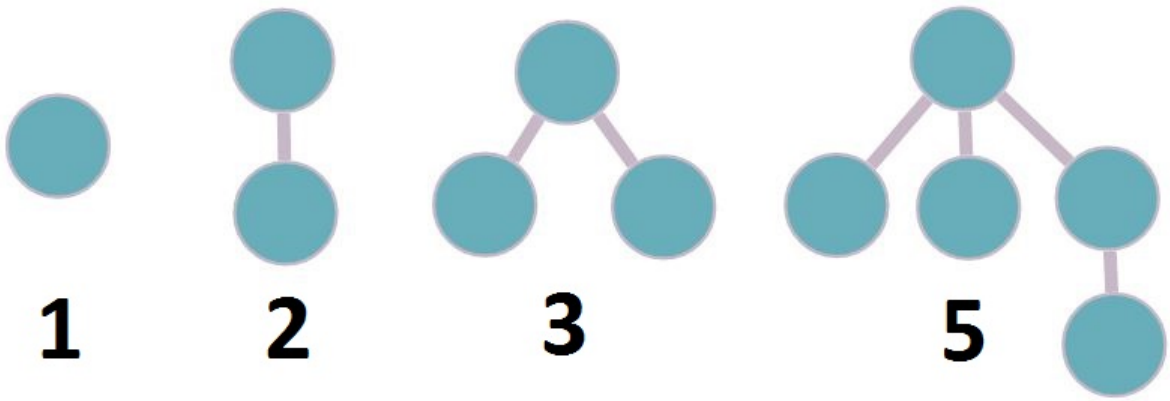
Slika 2.11: Prikaz $extractMin$ gdje se briše čvor 3, preuzeto iz [5]

Dokaz složenosti

Neka je x proizvoljan čvor u Fibonaccijevoj hrpi i pretpostavimo da je $deg(x) = k$. Neka su $y_1, y_2, ..., y_k$ oznake djece čvora x poredane kronološki s obzirom na trenutak povezivanja s x : y_1 je najranije povezan s x, a y_k posljednji. Tada je $deg(y_1) \geq$, $deg(y_i) \geq i - 2, \forall \in \{2, 3, ..., k\}$. Uzmemo li minimum za svaku moguću vrijednost i to bi izgledalo kao $\{0, 0, 1, 2, 3, ..., k - 4, k - 3, k - 2\}$.

Označimo sa $minVel(n)$ minimalnu moguću veličinu podstabla čvora stupnja n i dokažimo intuitivnu tvrdnju da vrijedi $minVel(n) \leq minVel(n + 1)$. Gledamo bilo kakvo stablo stupnja $n + 1$, $decreaseKey$ operacijom korijen stabla možemo otjerati u korijensku listu te nakon toga samo odrezati $n + 1$-i čvor još jednom $decreaseKey$ operacijom. Ovime smo pokazali da je iz bilo kojeg stabla stupnja $n + 1$ možemo legitimno iskonstruirati stablo stupnja n koje je strogo manje i to završava dokaz.

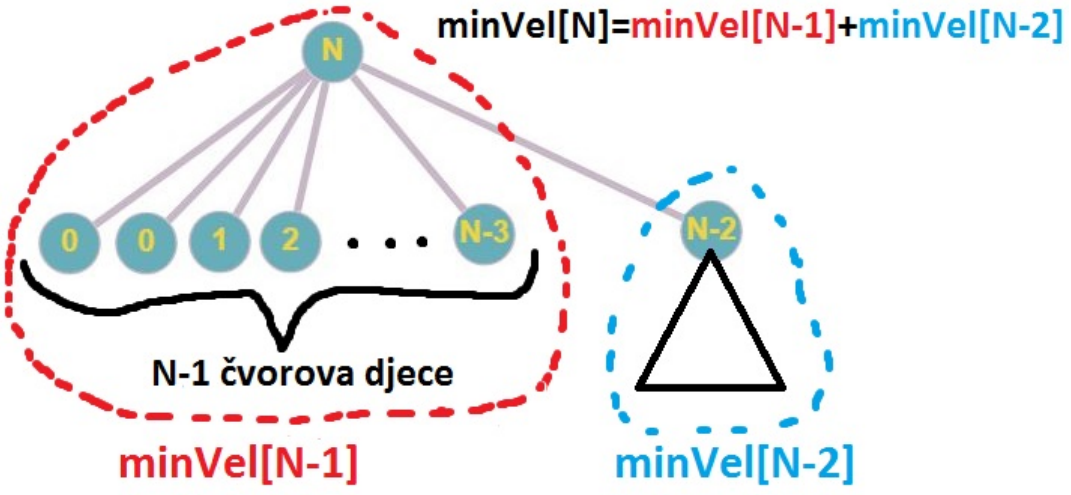
Iz ovih opservacija možemo zaključiti da za čvor stupnja k vrijedi $minVel(k) = minVel(0) + minVel(0) + minVel(1) + ...minvel(k - 2)$ jer je svaka druga kombinacija strogo veća. Sada samo trebamo nacrtati bazne slučajeve i vidjeti što se događa, prikaz 2.12. Možemo primijetiti da za ovih prvih par vrijedi $minVel(n)$=F_{n+2}, gdje je F_n n-ti Fibonaccijev broj[3].



Slika 2.12: S lijeva na desno minVel(0), minVel(1), minVel(2) i minVel(3).

Postavlja se: pitanje vrijedi li to općenito? Pogledajmo kako izgleda jedno podstablo sa N djece minimalne veličine na 2.13.

[3]Prisjetimo se: $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8$... i $F_n = F_{n-1} + F_{n-2}$



Slika 2.13: Izgleda podstabla s N djece minimalne veličine. Brojevi u čvorovima označavaju broj djece.

Primijetimo da se ovdje radi upravo o rekurzivnoj relaciji Fibonaccijevih brojeva. Sada imamo korak za bazu sa slike 2.12 i time dokazujemo da $min_vel(N) = F_{N+2}$, odakle i dolazi naziv Fibonaccijeva hrpa. Dakle podstablo Fibonaccijeve hrpe koje ima korijenski čvor stupnja d mora sadržavati minimalno F_{d+2} čvorova.

Na kraju treba razmotriti implikacije činjenice $min_vel(N) = F_{N+2}$. Ako imamo n čvorova u hrpi i za neki stupanj k vrijedi $n \leq F_{k+2}$ znači da je nemoguće da postoji u stablu čvor sa k ili više djece. Poznato je u literaturi [2] da vrijedi $F_n = \frac{\varphi^n}{\sqrt{5}} \pm 1$ gdje je $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$. Dakle ako uzmemo logaritam u bazi φ od našeg broja čvorova $\log_\varphi n$ imat ćemo gornju granicu broja djece u strukturi, a budući da se logaritmi međusobno razlikuju za koeficijent to je isto kao da imamo $koif * \log_2 n$ djece čime je dokaz gotov.

Logaritme je u implementaciji najlakše računati kao logaritme baze 2. Stoga ćemo izračunati logaritam kao $2 * \log_2 n$ jer to je jednako $\log_{\sqrt{2}} n = \log_{1.41} n$ što je veće od $\log_\varphi n$. Još se možemo dodatno osigurati da dodamo rezultatu $+3$ za svaki slučaj kako ne bismo podcijenili broj djece za 1 ili nešto.

2.2.5. Delete

Operacija $delete$ koja briše bilo koji čvor u hrpi se može ostvariti kombinacijom već postojećih operacija kao $decreaseKey(-beskonacno, info)$ + $extractMin$ što iznosi $O(1 + logn) = O(logn)$.

2.2.6. Objedinjujući teorem

Pažljivom čitatelju sljedeći teorem dolazi kao korolar svega do sad pročitanog:

Teorem: Složenost niza operacija nad Fibonaccijevom hrpom može se izraziti kao $O(\sum_{i=1}^{brojVrstaOperacija} brojOperacije_i * slozenostOperacije_i)$, gdje je složenost operacija $extractMin$ i $delete$ jednaka $O(logn)$ dok je složenost svih ostalih $O(1)$.

3. Performanse

Mjerit ćemo performanse Fibonaccijeve hrpe u usporedbi s binarnom hrpom. Primjeri koje ćemo isprobati su Dijkstrin algoritam u srednje i najgorem slučaju te neka generična situacija kada je puno $decreaseKey$ operacija i malo $extractMin$ operacija.

Računalo na kojem će se mjeriti performanse Dijkstre je Acer Aspire V5-552G s procesorom AMD A10-5757M APU. Procesor ima L1 i L2 cache memoriju od 16KB i 2MB tim redom, radne memorije ima 8GB DDR3. Računalo na kojem će se mjeriti "Malo ExtractMin, puno DecreaseKey operacija" dio ima procesor CPU RYZEN 5 5500U koje ima cache L3 memoriju iznosa 8 MB. Ovo računalo također ima 40GB DDR4 radne memorije što je bilo bitno jer Fibonaccijeva hrpa uzima puno memorije.

Informacije o cache memoriji bitne su za objašnjenje vremenskih skokova kada pređemo određenu razinu korištenje memorije u programu. Moja vlastita implementacija Fibonaccijeve hrpe nad kojom se provode mjerenja se nalazi na `https://github.com/Weramajstor/Seminar-2`.

3.1. Dijkstra

U uvodu je navedeno da Fibonaccijeva hrpa može spustiti složenost Dijkstre s $O((E + V)logV)$ na $O(E + VlogV)$. To je ostvarivo tako da svaki puta kada u Dijkstrinom algoritmom želimo unaprijediti put do nekog čvora koristimo operaciju $decreaseKey$ što je $O(1)$ umjesto da vadimo iz binarne hrpe prošli element i dodajemo novi od čega je svaka operacija $O(logn)$. Proučit ćemo neki prosječni nasumični slučaj i neki najgori slučaj u kojem se sigurno moraju ovako navedeno asimptotski izvoditi operacije te usporediti ponašanje Fibonaccijeve i binarne hrpe.

3.1.1. Prosječni slučaj

Generirat ćemo usmjereni graf od N čvorova u kojem svaki čvor ima točno $Stupanj$ izlaznih bridova prema drugim čvorovima, prema kojim će čvorovima ti bridovi ići kao

i njihove težine bit će generirane nasumično. Pogledajmo rezultate mjerenja u tablici 3.1.

Tablica 3.1: Vremena u sekundama izvođenja Dijkstre u nekim prosječnim slučajevima.

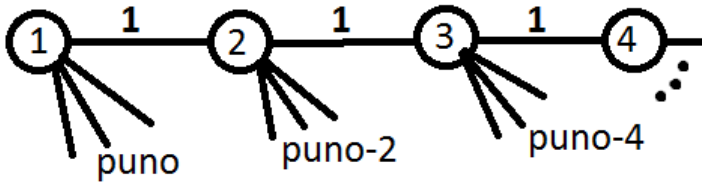
N	Stupanj	Fibonacci	Binarna	$\frac{Binarna}{Fibonacci}$
100	10	0.000763	0.000443	0.580603
100	50	0.000838	0.000604	0.720764
100	100	0.002009	0.00069	0.343454
1000	10	0.00268	0.001211	0.451866
1000	100	0.004618	0.00349	0.755738
1000	500	0.009602	0.008718	0.907936
1000	1000	0.015443	0.014948	0.967947
5000	10	0.012855	0.005762	0.44823
5000	100	0.023846	0.01703	0.714166
5000	1000	0.089991	0.089609	0.995755
5000	2500	0.196565	0.207688	1.05659
5000	5000	0.375995	0.382518	1.01735
10000	100	0.053146	0.035888	0.675272
10000	1000	0.194947	0.188056	0.964652
10000	5000	0.779011	0.800152	1.02714
10000	10000	1.5356	1.55352	1.01167
100000	10	0.476059	0.162182	0.340676
100000	100	0.827678	0.555551	0.671216
100000	1000	2.67891	2.49873	0.932742

Vidimo da je binarna hrpa daleko superiornija kao potpora Dijkstri na rijetkim grafovima. Na gustim grafovima ta se razlika smanjuje te je ponekad bolji i Fibonacci. Razlog zašto Fibonacci ne dominira na gustim grafovima je taj što zbog prisustva pregršta opcija već rano u algoritmu mnogi najbolji putevi budu relativno rano izračunati. Drugim riječima puno bridova neće biti uopće razmatrano kao moguće poboljšanje puta do nekog čvora.

Također razlog zašto je Fibonacci sporiji na rijetkim grafovima usprkos strogo bolje asimptotske složenosti je taj da ima visoku konstantu. Naime binarna hrpa se u implementaciji sastoji samo od jednog duljine N u kojem se lako penje elementi gore dolje dok smo već vidjeli da je Fibonaccijeva hrpa mnogo kompliciranija.

3.1.2. Najgori slučaj

Dijkstra izvodi najviše operacija u slučaju kada je graf potpun i mora za svaki brid koji proučava ažurirati udaljenosti do čvorova koje gleda. Kako bi izgledao takav slučaj možemo vidjeti na slici 3.1.



Slika 3.1: Izgleda podstabla s N djece minimalne veličine. Brojevi u čvorovima označavaju broj djece.

Tada se ostvaruju asimptotske složenosti $O((E+V)logV)$ za binarnu i $O(E+VlogV)$ za hrpu. Rastavimo složenost binarne $O((E+V)logV) = O(ElogV + VlogV)$ i uspoređivamo s Fibonaccijem $O(E+VlogV)$. Za potpun ili generalno gust graf, dominantni članovi u obje složenosti su $ElogV$ i E. Ovo bi značilo da bi se omjer $\frac{Binarna}{Fibonacci} = \frac{ElogV}{E} = logV$ trebao logaritamski rasti, pomnožen nekom konstantom.

Pogledajmo rezultate mjerenja prikazane tablicom 3.2. Broj čvorova N namjerno raste za faktor $\sqrt{2}$. Može se vidjeti da omjer otprilike raste logaritamski uz manje ili veće oscilacije zbog probijanja cache memorije, predviđanja grananja u sklopovlju nakon puno iteracija ili . Budući da je struktura grafa bila dosta eksplicitna nije bilo potrebno čuvati cijeli graf u memoriji već ga direktno generirati brid po brid i onda odmah zaboraviti na njega. Ovo je naglašeno jer bismo već za 10000 čvorova potrošili 400MB memorije samo za čuvanje grafa.

Dakle Fibonaccijeva hrpa kod Dijkstre je daleko superiornija u najgorem slučaju te bi ju imali smisla primijeniti u gustim grafovima ako želimo garanciju relativno brzog završetka naspram višednevnog vremena čekanja za imalo veće grafove. Međutim u većini primjena za koje je razumno pretpostaviti da su relativno nasumične brže je i dostupnije koristiti normalnu Dijkstru s binarnom hrpom, pogotovo za rijetke grafove. Usprkos svemu Fibonaccijeva Dijkstra daje važna teoretska asimptotska ograničenja algoritma pronalaska najkraćeg puta u težinskom grafu bez negativnih bridova. Na [3] se nalazi pregled složenosti algoritama za najkraći put, zanimljiv je Thorupov algoritam koji rješava ovaj problem nad cijelim brojevima u $O(E)$.

Tablica 3.2: Vremena izvođenja prethodnog opisanog najgoreg slučaja Dijsktre u sekundama.

N	Fibonacci	Binarna	$\frac{Binarna}{Fibonacci}$
20	1.1e-005	3.7e-005	3.36364
28	2.3e-005	8.8e-005	3.82609
39	2.2e-005	0.000107	4.86364
55	4e-005	0.000209	5.225
78	5.9e-005	0.000373	6.32203
110	0.000103	0.000756	7.33981
156	0.000228	0.001814	7.95614
220	0.000413	0.00433	10.4843
311	0.00077	0.007506	9.74805
439	0.001605	0.015212	9.47788
620	0.003304	0.032071	9.70672
876	0.005334	0.066755	12.515
1236	0.011335	0.127203	11.2221
1751	0.02105	0.274411	13.0362
2472	0.042698	0.568606	13.3169
3496	0.088064	1.14583	13.0113
4944	0.16225	2.34924	14.4791
6986	0.308252	5.41779	17.5759
9888	0.63945	11.9873	18.7463
13972	0.86119	17.2143	19.989
19777	2.64361	50.9526	19.2739
27945	5.24628	112.45	21.4343
39554	11.1553	258.886	23.2074
55909	24.0201	553.129	23.0278
79008	45.4244	1091.97	24.0392
111818	86.6513	2445.41	28.2213
158016	163.664	4766.52 (1h 19.5min)	29.1237
223636	375.257	9947.58 (2h 46min)	26.5087
316032	742.222	20300.7 (5h 38min)	27.3512

3.2. Malo ExtractMin, puno DecreaseKey operacija

Ovakva situacija se može naći kod već uvodno navedenog "trendinga" na društvenim mrežama. Neke daljnje teoretske primjene bi bile: procesi i prioriteti u operacijskom sustavu, nadmetanje u online aukcijama gdje bi zapravo imali max-heap jer ponude rastu, min-heap za mijenjanje količine inventara po skladištima dućana jer se količine dobara smanjuju dok se ne naruči novo i razni drugi primjeri.

Konkretno mjerit će se slučaj kada je 100 $decreaseKey$ operacija za svaku $extractMin$ operaciju. Uz vrijeme izvođenja sad će se prikazati i korištenje memorije.

Tablica 3.3: Izvođenje 1123123 operacija od kojih je svaka stota extractMin.

N	$2^N - 1$	Fibonacci/s	Fibonacci/mem	Binarna/s	Binarna/mem
1	1	0.00232	80B	0.00223	4B
2	3	0.0057	160B	0.00474	12B
3	7	0.00395	320B	0.00655	28B
4	15	0.00334	750B	0.00807	60B
5	31	0.00331	1.5KB	0.00942	124B
6	63	0.00383	3KB	0.01106	252B
7	127	0.00528	6KB	0.01224	508B
8	255	0.00817	12.5KB	0.01487	1KB
9	511	0.01351	25KB	0.01529	2KB
10	1023	0.02285	50KB	0.01682	4KB
11	2047	0.03163	100KB	0.01899	8KB
12	4095	0.03761	200KB	0.02069	16KB
13	8191	0.04369	400KB	0.02157	32KB
14	16383	0.05186	800KB	0.02352	64KB
15	32767	0.05842	1.56MB	0.02506	128KB
16	65535	0.06325	3.125MB	0.02657	256KB
17	131071	0.12376	6.25MB	0.0283	512KB
18	262143	0.16636	12.5MB	0.03036	1MB
19	524287	0.1848	25MB	0.03276	2MB
20	1048575	0.1841	50MB	0.04103	4MB
21	2097151	0.1839	100MB	0.0794	8MB
22	4194303	0.189	200MB	0.11394	16MB
23	8388607	0.1760	400MB	0.15534	32MB
24	16777215	0.1674	800MB	0.181	64MB
25	33554431	0.1573	1.68GB	0.226	128MB
26	67108863	0.1574	3.2GB	0.2854	256MB
27	134217727	0.1608	6.4GB	0.3584	512MB
28	268435455	0.1578	12.8GB	0.4194	1GB
29	536870911	0.1517	25.6GB	0.4773	2GB

Uvijek izvodimo 1123123 operacija i stoga bi kada bi pristup memoriji jednako koštao neovisno o tome koliko je koristimo onda bi Fibonaccijeva hrpa imala konstantnu

složenost za bilo koji N. U početku kada sve stane u niži L1 cache ili čak procesorske registre vidimo da je Fibonacci hrpa bolja sve do trenutka N=10, zatim raste dalje sve do trenutka N=18 kada zauzme više od 8MB L3 cache-a i dalje je neovisno o korištenju memorije jednake brzine.

Za binarnu hrpu vidimo da konstantno raste te je lošija na početku i na od N=24 na dalje. Možemo lijepo vidjeti kako logaritamski raste jer otkad je prošla 8MB vrijeme izvođenja za svaki novi N naraste za +0.03 do +0.07. Iz ovog svega vidimo da je Fibonaccijeva hrpa primjenjivo bolja za veliki broj čvorova.

4. Zaključak

Postavlja se pitanje je li za primjere iz pravog života pa onda i ove dovoljna samo binarna hrpa ili postoji specifično rješenje koje bolje rješava problem od hrpa. Odgovor je uglavnom da je zbog visokih memorijskih zahtjeva Fibonaccijeve hrpe što između ostalog implicira spor početak dok ne alocira masivne količine memorije. Dodatna otežavajuća okolnost je da su operacije Fibonaccijeve hrpe amortizirane(postoji teoretski neamortizirana hrpa s istim složenostima operacija [1]) odnosno pojedinačno mogu doseći složenost $O(n)$ što ju čini manje primjenjivom kod sustava u realnom vremenu (eng. Real-time systems).

Sve u svemu Fibonaccijeva hrpa ima nekolicinu primjena i od teoretskog je značaja za određivanje složenosti algoritama.

5. Literatura

[1] URL https://en.wikipedia.org/wiki/Brodal_queue. Pristupljeno 5.5.2023.

[2] URL https://en.wikipedia.org/wiki/Fibonacci_sequence#Relation_to_the_golden_ratio. Pristupljeno 21.4.2023.

[3] URL https://en.wikipedia.org/wiki/Shortest_path_problem#Undirected_graphs. Pristupljeno 5.5.2023.

[4] URL https://en.wikipedia.org/wiki/Fibonacci_heap. Pristupljeno 21.4.2023.

[5] T. Cormen; C. Leiserson; R. Rivest; C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[6] M. Fredman; R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal ofthe Association for Computing Machinery*, 34(3): 596–615, 1987.

6. Sažetak

Opisuje se i dokazuje struktura Fibonaccijeve hrpe. Navode se algoritmi Dijkstra i Prim čija je nadogradnja motivacija korištenja Fibonaccijeve hrpe. Pruža se implementacija u C++u te se mjeri performanse na gustim i rijetkim grafovima. Razmatraju se mogućnosti primjene Fibonaccijeve hrpe i na druge probleme.