Fibonaccijeva hrpa

Lončar, Marino

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Marino Lončar

FIBONACCIJEVA HRPA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Saša Singer

Zagreb, veljača, 2016.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Kombinatorna optimizacija je područje matematike koje se bavi pronalaskom optimalnih objekata iz konačnog skupa objekata. Jedan od najpoznatijih problema kombinatorne optimizacije je problem najkraćeg puta, koji se može općenito definirati kao problem pronalaska optimalnog puta od točke A do točke B. S ovim problemom se svakodnevno susreću razni povezani sustavi, kao što su internet, društvene mreže i cestovne mreže. Prilikom slanja internetskih paketa, potrebno je pronaći najkraći broj računala koji paket mora proći od početnog računala do odredišta. Rješenje ovog problema u optimalnom vremenu omogućava brz i pouzdan rad cijelog interneta. Problem minimalnog puta u prometu se najbolje očituje u navigacijskim uređajima koji u roku od par sekundi računaju najkraće puteve od jednog mjesta do drugog. Pouzdano rješenje za ovaj problem je nužno zbog velike popularnosti grafova u matematičkom modeliranju sustava pri čemu je najkraći put između dva vrha jedno od najtraženijih "objekata" u grafu.

S obzirom da je ovaj problem odavno poznat, dosad su se razvila mnoga rješenja od kojih je, zasigurno najpopularniji Dijkstrin algoritam. Dijkstrin algoritam pretražuje vrhove koji imaju najviše "potencijala" da budu na optimalnom putu do odredišta. Kako bi Dijkstrin algoritam efikasno riješio problem najkraćeg puta, oslanja se na pomoć strukture koja će jednostavno moći iz skupa vrhova dohvatiti one koje je prvo potrebno pretražiti.

Upravo opisane strukture su i tema ovoga rada. Rad je podijeljen u 4 poglavlja. U prvome poglavlju opisat će se jedna apstraktna struktura i jedna njezina jednostavna implementacija. U drugom poglavlju bit će predstavljena i opisana efikasnija implementacija apstraktne strukture čija će primjena biti objašnjena u trećem dijelu rada. U završnom poglavlju se uspoređuju efikasnosti opisanih struktura pomoću algoritama koji ih koriste.

Poglavlje 1

Hrpa

1.1 Prioritetni red

Prioritetni red (engl. Priority Queue) je apstraktna struktura podataka u kojoj se svakom spremljenom elementu pridružuje dodatna vrijednost zvana prioritet (ili ključ). Najvažnijim elementima se najčešće pridružuje manji prioritet, pa se, za razliku od standardnog reda, prilikom dohvaćanja elementa iz reda ne vraća prvi element ubačen u red, već onaj s najmanjim prioritetom. Na prioritetnom redu se definiraju sljedeće operacije:

- $insert(x, P)$ – ubacuje novi element x s predefinirim prioritem u red P.
- $findMin(P)$ – vraća se element s najmanjim prioritetom iz prioritetnog reda P.
- $deleteMin(P)$ – iz prioritetnog reda P izbacuje element s najmanjim prioritetom i vraća izbačeni element.
- $decreaseKey(x, k, P)$ – u prioritetnom redu P smanjuje vrijednost prioriteta elementa x na k.

Također se zahtijeva da je na skupu prioriteta definiran totalni uređaj.

Primjene prioritetnog reda pojavljuju se u mnogim segmentima računarstva, kao što su internetski promet i operacijski sustavi. U svakoj sekundi tisuće paketa prolaze usmjernicima (engl. router), prilikom čega svi paketi nemaju isti prioritet, jer su neki bitniji za ispravno funkcioniranje programa kojima su namijenjeni, kao što su primjerice VOIP programi. Unutar operacijskog sustava postoji mnogo procesa koji čekaju na procesorsko vrijeme. Neki od njih su važni za ispravno funkcioniranje cijelog sustava, pa se procesima pridjeljuju različiti prioriteti, pri čemu najbitniji procesi dobivaju najmanji prioritet. Kada se oslobode resursi, operacijski sustav dodjeljuje resurse sljedećem najvažnijem procesu.

Prioritetni red je moguće implementirati na više načina, među kojima su sortirana vezana lista, binarno stablo traženja i hrpa. U sortiranoj vezanoj listi, najmanji element se nalazi na prvom mjestu liste, stoga je vremenska složenost operacija *deleteMin()* i *findMin()* $O(1)$. Vremenska složenost operacije *insert()* je $O(n)$ jer je potrebno iterirati po listi da bi se pronašlo odgovarajuće mjesto novog elementa. Operacija *decreaseKey()* zahtijeva $O(n)$ vremena jer je potrebno pronaći element u listi, izbaciti ga i smjestiti na pravo mjesto, ovisno o novom prioritetu. Kod implementacije pomoću binarnog stabla traženja, sve 4 navedene operacije imaju složenost $O(\log n)$, ukoliko je stablo balansirano. Implementacija pomoću hrpe pokazuje se kao jedna od najefikasnijih i najčešćih, pa se zbog toga često u literaturi pojmovi hrpe i prioritetnog reda poistovjećuju. Stoga će se u ovom radu prioritetnim redom smatrati apstraktna struktura podataka, a hrpom jedna moguća implementacija prioritetnog reda.

1.2 Hrpa

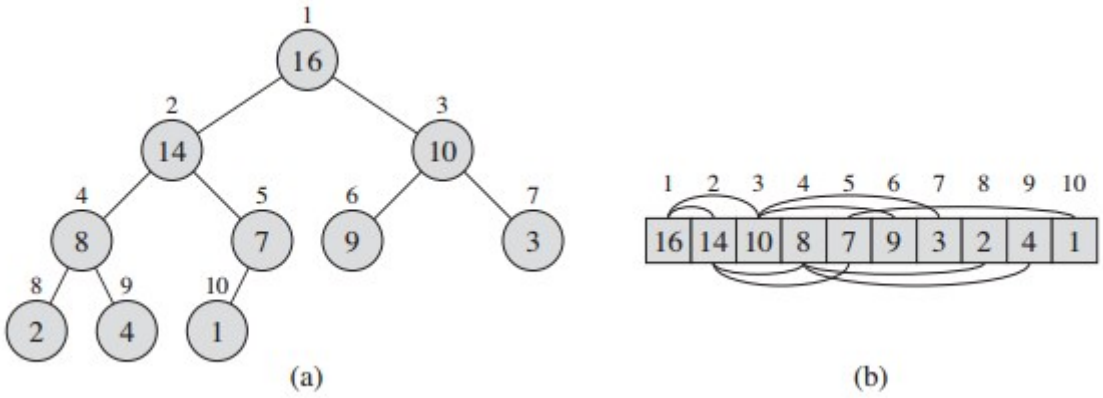
Definicija 1.2.1. *Potpuno binarno stablo T je **(binarna) hrpa** ako su ispunjeni sljedeći uvjeti:*

- *Čvorovi stabla T su označeni podacima tipa na kojem je definiran totalni uređaj.*
- ***(Svojstvo hrpe)** Neka je i bilo koji čvor stabla T. Tada je oznaka čvora i manja ili jednaka od oznake bilo kojeg njegovog djeteta.*

Hrpa je potpuno binarno stablo, što znači da su svi nivoi stabla popunjeni, osim možda posljednjeg koji se puni ”slijeva nadesno”. Za hrpu nije nužno da je binarno stablo, već može biti stablo s proizvoljnim brojem djece, pa tada govorimo o **d-hrpama**, pri čemu je d broj djece čvora u stablu. Iz definicije je jasno da se element s minimalnom oznakom uvijek nalazi u korijenu stabla. Postoje dvije vrste hrpa: **min-hrpa** i **max-hrpa**. Hrpa opisana u prethodnoj definiciji je min-hrpa, dok se kod max-hrpe najveći element nalazi u korijenu i svaki čvor ima oznaku veću ili jednaku od oznaka svoje djece. U nastavku ovog rada, ako nije drugačije navedeno, hrpom će se smatrati smatrati min-hrpa. Jedan primjer hrpe dan je na slici 1.1. Iz primjera i definicije se vidi da ne postoji nikakva relacija između elemenata na istom nivou stabla.

Pojmove koji se vežu uz stablo možemo prenijeti i na hrpu. Tada je visina čvora u hrpi duljina najdužeg puta od čvora do nekog lista u stablu, a visina hrpe se definira kao visina korijena hrpe. Hrpa ima oblik potpunog binarnog stabla, pa je poznato da je visina hrpe koja sadrži n elemenata jednaka $\lfloor \log n \rfloor$. Ova činjenica će biti bitna u analizi vremenske složenosti operacija na hrpi.

Potpuno binarno stablo, pa tako i hrpa, se na računalu može prikazati na 2 načina, pomoću pokazivača i pomoću polja. U prikazu pomoću pokazivača, svaki element stabla



Slika 1.1: Primjer jedne max-hrpe. a) Hrpa u obliku binarnog stabla, b) Hrpa prikazana poljem, strelice između elemenata označuju relaciju roditelj-dijete.

ima pokazivače na svog roditelja, te lijevo i desno dijete, a stablo se može tada prikazati pokazivačem na korijen stabla. Reprezentira li se stablo poljem, svaki element je spremljen u jednom mjestu polja i identificiran indeksom mjesta na kojem je spremljen. Korijen stabla se nalazi na prvom mjestu, a ako je i indeks elementa u polju, tada se lako mogu dohvatiti indeksi roditelja, lijevog brata i desnog brata na sljedeći način:

- $roditelj(i) = \lfloor i/2 \rfloor$
- $lijevi(i) = 2i$
- $desni(i) = 2i + 1$

Ove operacije se na većini računala mogu vrlo efikasno implementirati operacijama pomicanja bitova. Roditelj čvora i se može dobiti pomicanjem bitova čvora i za jedan bit udesno, a njegovo lijevo dijete se dobije pomicanjem bitova čvora i za jedan bit ulijevo. Iako oba prikaza imaju linearnu prostornu složenost, prikaz pomoću polja se pokazao praktičnijim, posebice u implementaciji *heapsort* algoritma koji će biti opisan u nastavku.

Operacije na hrpi

Nad hrpom su definirane sljedeće operacije:

- $makeHeap()$ – stvara i vraća novu praznu hrpu.
- $insert(x, h)$ – ubacuje novi element x u hrpu h.
- $deleteMin(h)$ – izbacuje i vraća najmanji element iz hrpe h.

- $findMin(h)$ – vraća najmanji element iz hrpe h.
- $decreaseKey(x, k, h)$ – elementu x u hrpi h smanjuje vrijednost za k.
- $minHeapify(x, h)$ – za element x u hrpi h popravlja svojstvo hrpe (ako je narušeno) u podstablu kojem je x korijen. Pretpostavka je da je svojstvo hrpe zadovoljeno u lijevom i desnom podstablu od x.
- $buildMinHeap(A)$ – iz danog polja A, stvara hrpu.

Operacija *decreaseKey()* se ne pojavljuje u standardnim implementacijama hrpe, ali je potrebna, ako se želi hrpom implementirati prioritetni red. Operacija *buildMinHeap()* također nije standardni dio hrpe, no potrebna je za heapsort algoritam. Ukoliko se radi o max-hrpi, tada pripadne operacije zamjenjujemo s *deleteMax()*, *findMax()*, *increaseKey()*, *maxHeapify()* i *buildMaxHeap()*.

Stvaranje hrpe je trivijalna operacija jer je potrebno inicijalizirati novo prazno stablo, stoga je složenost operacije *makeHeap()* $O(1)$. Operacija *findMin()* je, također, izvediva u konstantnom vremenu jer je dovoljno pročitati vrijednost iz korijena hrpe. Ubacivanje novog elementa u hrpu se obavlja u dva koraka. Prvo se ubacuje novi element x u hrpu na prvo slobodno mjesto na zadnjem nivou hrpe. Ovim korakom je vjerojatno narušeno svojstvo hrpe, pa je potrebno dovesti x na pravo mjesto u hrpi, tako što se x zamjenjuje sa svojim roditeljem sve dok x ne postane korijen hrpe, ili dok vrijednost x ne bude veća od vrijednosti njegovog roditelja. Složenost ubacivanja je $O(1)$, dok je za dovođenje novog elementa potrebno najviše $O(\log n)$ koraka, jer je broj zamjena ograničen visinom hrpe, pa je složenost operacije *insert()* $O(\log n)$.

Operacija *decreaseKey()* se obavlja na sličan način kao i ubacivanje novog elementa, tako što se traženom elementu smanji vrijednost i zamjenjuje s roditeljem dok se ne obnovi svojstvo hrpe. Potrebno je naglasiti da hrpa ne podržava efikasan način traženja proizvoljnog elementa u hrpi, jer je potrebno pretražiti cijelo stablo što zahtijeva $O(n)$ operacija. Kako bi ova operacija bila efikasna, potrebno je imati dodatnu strukturu kojom će biti moguće, u konstantnom vremenu, za svaki element, pronaći njegovo mjesto u hrpi. Druga mogućnost je da elementi spremljeni u hrpi, koja implementira prioritetni red, imaju spremljenu svoju poziciju u hrpi. Ako se pretpostavi da je moguće pronaći element x u hrpi u konstantnom vremenu, tada operacija *decreaseKey()* ima vremensku složenost u najgorem slučaju $O(\log n)$.

Operacija *minHeapify()* služi kako bismo popravili svojstvo hrpe ukoliko se naruši prilikom promjena u hrpi. Neka je x element takav da je veći od barem jednog svog djeteta i neka je svojstvo hrpe valjano u lijevom i desnom podstablu od x. Tada se x uspoređuje sa svojom djecom i zamjenjuje mjesto s manjim od svoje djece. Rekurzivnim pozivom operacije *minHeapify()* na x, obnavlja se svojstvo hrpe u podstablu kojem je x korijen. Ovim postupkom će x postati list hrpe, ili će biti manji ili jednak od svoje djece, čime će svojstvo

hrpe biti obnovljeno. Pseudokod ove operacije na hrpi, prikazanoj poljem, dan je na slici 1.2. Zamjena x s nekim svojim djetetom je moguća u $O(1)$, dok je vrijeme izvršavanja rekurzivnih poziva $O(\log n)$ [8], pa je vremenska složenost *minHeapify()* $O(\log n)$. Preciznije, ukoliko je visina elementa u hrpi h tada je vrijeme izvršavanja *minHeapify()* na njemu $O(h)$.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Slika 1.2: Pseudokod operacije *maxHeapify()* na max-hrpi prikazanoj poljem A. Želi li se dobiti *minHeapify()* operacija, dovoljno je promijeniti znak > u <.

Brisanje minimalnog elementa je jednostavno iskoristi li se prethodno opisana operacija. Prvo se zapamti vrijednost koju je potrebno vratiti i ukloni se korijen hrpe. Ovim korakom je narušena struktura stabla, pa se za korijen hrpe postavi posljednji element na zadnjem nivou hrpe. Svojstvo hrpe je narušeno nakon ovog koraka, stoga se pozivom *minHeapify()* na korijenu stabla popravi struktura hrpe. Zamjena minimalnog i posljednjeg elementa zahtijeva $O(1)$ operacija, a poziv *minHeapify()* $O(\log n)$ operacija, pa je složenost *deleteMin()* operacije $O(\log n)$.

Polje, dano kao ulaz, u operaciji *buildMinHeap(A)* može se promatrati kao hrpa prikazana poljem. Raspored elemenata u polju A je proizvoljan, pa svojstvo hrpe ne mora biti zadovoljeno. Ideja je preurediti polje tako da novi poredak elementa predstavlja valjanu hrpu. To se postiže građenjem hrpe od dna prema vrhu pomoću *minHeapify()* operacije. Počinje se od zadnjeg nivoa i na svakom elementu nivoa se pozove *minHeapify()* operacija. U sljedećem koraku se ista operacija poziva na svakom elementu višeg nivoa. Nakon i-tog koraka, sva podstabla visine i ili manje su valjane hrpe. Nakon $O(\log n)$ koraka će ulazno polje predstavljati valjanu hrpu. Iako izgleda da ovaj algoritam zahtijeva ukupno $O(n \log n)$ operacija, jer se na svakom elementu jednom poziva *minHeapify()*, pokazuje se da je složenost ove operacije $O(n)$ [8] jer vrijeme izvođenja *minHeapify()* ovisi o visini čvora na kojem se operacija poziva.

Iako hrpa ima izvrsne složenosti osnovnih operacija, postavlja se pitanje poboljšanja osnovnih operacija. Od prvog pojavljivanja hrpe 1964. godine, pojavile su se razne verzije hrpa, često mnogo složenije za implementaciju, koje su imale bolja asimptotska vremena izvršavanja operacija. Jedna od njih, Fibonaccijeva hrpa, biti će detaljnije opisana u sljedećem poglavlju.

1.3 Primjene hrpe

Hrpa ima dvije primarne primjene u računarstvu. Naime, hrpa je ključan element u efikasnom algoritmu sortiranja zvanom *heapsort*. Druga i raširenija primjena hrpe je u implementaciji prioritetnog reda koji je nužan dio algoritama i problema koji će biti opisani.

Heapsort

Definicija 1.3.1. *Neka je* $A = [x_1, .., x_n]$ *proizvoljni niz duljine n. Problem sortiranja niza je problem preuređivanja niza tako da članovi niza budu:*

- *uzlazno poredani:* $x_1 \leq x_2 \leq \ldots \leq x_n$ *, ili*
- *silazno poredani:* $x_1 \geq x_2 \geq \ldots \geq x_n$.

Sortiranost niza u računarstvu je uvijek poželjna iz razloga što mnogi algoritmi imaju uvjet da je niz na kojem rade sortiran. Jedan od najpoznatijih i najjednostavnijih algoritama, binarno pretraživanje, zahtijeva da niz bude sortiran kako bi se traženi element uspješno pronašao. Neki jednostavni algoritmi kao *insertion sort* i *selection sort* sortiranje niza od n elemenata obavljaju u vremenu $O(n^2)$, dok najpopularniji algoritmi u praksi, *quicksort* i *mergesort*, sortiraju niz u $O(n \log n)$ vremenu (u prosječnom slučaju).

Heapsort algoritam predstavio je Williams 1964. godine [11]. Ideja algoritma je jednostavna i temelji se na korištenju hrpe za sortiranje niza. U prvom koraku algoritma se svi elementi niza ubace u hrpu, pri čemu se zbog svojstva hrpe, najmanji element nalazi u korijenu hrpe. Zatim se n puta briše najmanji element hrpe, kojim se puni niz ponovno od početka prema kraju. Na kraju algoritma, hrpa će biti prazna, a niz uzlazno sortiran. Opisani algoritam zahtijeva n operacija *insert()* kojima je ukupno vrijeme izvršavanja je $O(n \log n)$ i n operacija *deleteMin()* s ukupnim vremenom izvršavanja $O(n \log n)$. Ukupna složenost algoritma *heapsort* je $O(n \log n)$, što ga čini, u prosječnom slučaju, teoretski jednako vremenski efikasnim kao *quicksort* i *mergesort*. U ovoj implementaciji prostorna složenost je $O(n)$, jer je potrebna hrpa kao dodatna pomoćna struktura. Vremenska složenost algoritma se ne može poboljšati, no prostorna složenost se može svesti na $O(1)$.

Floyd je 1964. godine [5] predstavio poboljšanje *heapsort* algoritma, koje sortira niz u mjestu, tj. s prostornom složenosti $O(1)$. Memorija koja se koristi za spremanje niza,

koristi se i za kreiranje hrpe. Za razliku od prethodne varijante algoritma, u ovoj se koristi max-hrpa. Algoritam započinje pozivanjem operacije *buildMaxHeap(A)* koja u vremenu $O(n)$ preuredi poredak elemenata u nizu, tako da niz predstavlja valjanu hrpu. Tim korakom se najveći element smjesti na prvo mjesto, nakon čega se izbaci najveći element tako da se zamijene prvi i zadnji element hrpe, a veličina hrpe se smanji za 1. Opisanom zamjenom najveći element se smjesti na posljednje mjesto u nizu. Nakon toga se operacijom *maxHeapify()* hrpa ponovno dovede u valjano stanje. Ponavljanjem ovih koraka n puta, niz će biti silazno sortiran, jer se u svakom koraku na zadnje mjesto hrpe postavlja najveći trenutni element i smanjuje hrpu. Primjer rada ove varijante *heapsort* algoritma je dan na slici 1.3.

Slika 1.3: Primjer rada *heapsort* algoritma u mjestu. Prvi dio algoritma je pretvaranje niza u hrpu, dok je u drugom dijelu algoritma, crnim obrubom označen maksimalni element izbačen iz hrpe u tom koraku.

Usprkos izvrsnom teoretskom vremenu izvršavanja i mogućnosti sortiranja u mjestu, *heapsort* nije uspio u praksi nadmašiti popularnost *quicksort*-a i *mergesort*-a. Prvi problem *heapsort*-a je što ne omogućava stabilno sortiranje, tj. ne čuva relativan poredak elemenata s istim vrijednostima nakon sortiranja, što je moguće postići s *mergesort*-om. Drugi problem se javlja zbog male memorije procesora, pa se često ne može učitati cijeli niz u memoriju. Zbog toga usporedba elemenata u hrpi može trajati duže od očekivanog, jer se roditelj i njegova djeca ne nalaze jedni do drugih u memoriji. Iz ovih razloga će se *heapsort* rijetko naći implementiran u standardnim biblitekama popularnih programskih jezika.

Hrpa se također može koristiti i u problemu parcijalnog sortiranja, kod kojega se traži da je samo prvih m elemenata niza sortirano. Ovaj problem moguće je riješiti popunjavanjem hrpe danim nizom i zatim m puta dohvatiti minimalni element hrpe. Ovo rješenje ima vremensku složenost $O(n + m \log n)$.

Algoritmi selektiranja

Algoritam selektiranja je algoritam za određivanje najmanjeg, najvećeg ili k-tog najmanjeg elementa niza. Da bismo pronašli najmanji ili najveći element, dovoljno je elemente ubaciti u hrpu i tada u vremenu $O(1)$ dohvatiti najmanji ili najveći element, po potrebi. Ukoliko se traži k-ti najmanji element niza, potrebno je k puta izbaciti najmanji element hrpe.

Ponekad je potrebno vratiti prvih k elemenata niza. Primjer ovakvog problema su internetski upiti kod kojih je korisniku potrebno vratiti određeni broj odgovora na upite. Problem je moguće riješiti korištenjem max-hrpe, tako što se na početku algoritma hrpa popuni s prvih k elemenata niza. Zatim se svaki sljedeći element niza ubaci u hrpu, a iz hrpe se izbaci najveći element. Ovime se osigurava da će u hrpi, u svakom trenutku algoritma, biti najmanjih k pročitanih elemenata niza. Kada algoritam završi, hrpa će sadržavati najmanjih k elemenata početnog niza. Algoritam se izvodi u vremenu $O(n \log k)$ i pogodan je za slučajeve kad je k relativno malen.

Simulacije temeljene na događajima

Programi koji simuliraju kretanje n čestica u nekom sustavu moraju često provjeravati moguće međusobne sudare čestica, koje utječu na njihova kretanja. Simulaciju je moguće obnavljati u nekom pravilnom vremenskom intervalu, no tada se moraju provjeriti uvjeti svih mogućih n^2 kolizija, na kraju svakog intervala, i uz to je potrebno odrediti vremenski poredak tih sudara, kako bi se simulacija mogla ispravno obavljati. Provjera kolizija se može poboljšati na način da se simulacija ne izvodi u pravilnim vremenskim intervalima, već samo na događajima koji mijenjaju uvjete u sustavu. Unaprijed se mogu odrediti budući sudari, koji se spremaju u prioritetni red implementiran hrpom, pri čemu se svakom sudaru kao ključ prirodaje vrijeme do sudara. Tim postupkom je poznato vrijeme sljedećeg događaja kada se simulacija mora ponovno računati i osigurava se sustav od nepotrebnog računanja kada nema promjena u sustavu.

Algoritmi na grafovima

Prioritetni redovi se javljaju kao prirodna pomoćna struktura u mnogim algoritmima na grafovima, kao što su Dijkstrin algoritam, Primov algoritam pronalaženja minimalnog razapinjućeg stabla i A^* algoritam. Ovi algoritmi zahtijevaju pogodnu pomoćnu strukturu za spremanje vrhova zajedno s njihovim prioritetima. Što je veći prioritet vrha, potrebno ga

je prije posjetiti. Hrpa se ponovno pokazuje kao efikasna implementacija prioritetnog reda za Dijsktrin i Primov algoritam koji će biti detaljnije opisani u sljedećim poglavljima.

Poglavlje 2

Fibonaccijeva hrpa

Fibonaccijevu hrpu opisali su Fredman i Tarjan 1984. godine u [6], s idejom predstavljanja efikasnije implementacije prioritetnog reda kojom bi se poboljšalo asimptotsko vrijeme izvođenja najpopularnijih optimizacijskih algoritama na grafovima. Efikasnost Fibonaccijeve hrpe se postiže metodom "lijenog izračunavanja", tj. odgađanjem obavljanja vremenski "skupih" operacija sve dok nisu nužno potrebne. Fibonaccijeva hrpa je proširenje binomnog reda, varijante hrpe koju je 1978. godine predstavio Vuillemin u [10]. Kod binomnog reda, sve osnovne operacije na hrpi, uključujući i spajanje dviju hrpa, imaju, u najgorem slučaju, vremensku složenost $O(\log n)$.

2.1 Struktura Fibonaccijeve hrpe

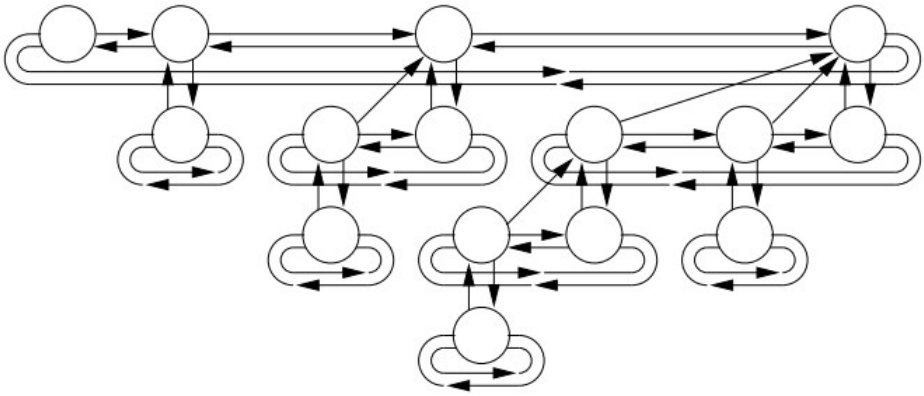
Fibonaccijeva hrpa (ili kraće **F-hrpa**) je skup stabala koja zadovoljavaju svojstvo hrpe (oznaka djeteta je veća ili jednaka oznaci roditelja). Element F-hrpe je reprezentiran čvorom u stablu. Sva stabla su međusobno disjuktna, što znači da se svaki element F-hrpe nalazi u točno jednom stablu. Ne postavljaju se nikakva eksplicitna ograničenja na strukturu i broj stabala, već će se ograničenja implicitno manifestirati zbog načina manipuliranja stablima. Stabla su povezana tako što se svi korijeni stabala nalaze u dvostruko vezanoj cirkularnoj listi zvanoj *lista korijena* (engl. root list). F-hrpa je reprezentirana pokazivačem na najmanji korijen svih stabala, tj. na najmanji element F-hrpe.

Svaki čvor x u F-hrpi ima sljedeće vrijednosti:

- *x.parent* – pokazivač na roditelja čvora x. Ako je x korijen stabla, tada je vrijednost *parent(x)* jednaka *null*.
- *x.left* – pokazivač na lijevog brata čvora x. Ako x nema braće, tada *x.left* pokazuje na x.

- *x.right* – pokazivač na desnog brata čvora *x*. Ako *x* nema braće, tada *x.right* pokazuje na *x*.
- *x.child* – pokazivač na proizvoljno dijete čvora *x*. Ako *x* nema djece, *x.child* ima vrijednost *null*.
- *x.rank* – *rang* (ili *stupanj*) čvora *x* je broj njegove djece. Ne postoji eksplicitno ograničenje na broj djece proizvoljnog čvora u F-hrpi.
- *x.marked* – ako je vrijednost *x.marked* jednaka *true*, kažemo da je čvor *x označen*, u suprotnom da je *x neoznačen*. Korijen stabla je uvijek neoznačen. Uloga ove vrijednosti bit će pokazana prilikom opisa operacija na F-hrpi.
- *x.key* – ključ/prioritet elementa spremljenog u čvoru *x* F-hrpe.

Prikaz jedne F-hrpe dan je na slici 2.1. Djeca svakog čvora su, kao i korijeni stabala, povezani u dvostruko vezanoj cirkularnoj listi. Razlog korištenja dvostruko vezanih cirkularnih lista je što omogućavaju spajanje dviju lista, brisanje elementa i dodavanje elementa u listu u konstantnom vremenu. Reprezentiranje prazne F-hrpe se obavlja postavljanjem minimalnog čvora F-hrpe na *null*. Iz opisa strukture F-hrpe očito je da F-hrpa s n spremljenih elemenata ima prostornu složenost $O(n)$.



Slika 2.1: Primjer izgleda jedne Fibonaccijeve hrpe, pri čemu *null* pokazivači nisu prikazani zbog preglednosti prikaza hrpe.

2.2 Operacije na Fibonaccijevoj hrpi

Na Fibonaccijevoj hrpi H definirane su sljedeće operacije:

- *makeHeap()* – stvara i vraća novu F-hrpu.
- *findMin(H)* – vraća najmanji element spremljen u F-hrpi H.
- *deleteMin(H)* – briše najmanji element iz F-hrpe H i vraća izbrisani element.
- *decreaseKey(k,x,H)* – smanjuje vrijednost ključa elementa x u F-hrpi H za k.
- *meld(H_1, H_2)* – spaja F-hrpe H_1 i H_2 u novu F-hrpu i vraća hrpu nastalu spajanjem H_1 i H_2.
- *delete(x,H)* – briše element x iz F-hrpe H.

Za proizvoljni element x u F-hrpi, kao i u standardnoj hrpi, ne postoji efikasan način pronalaska čvora u kojem je spremljen x. Operacije *decreaseKey(k,x,H)* i *delete(x,H)*, za efikasno obavljanje svog zadatka, zahtijevaju pristup čvoru s elementom x u $O(1)$ vremenu. Jedno moguće rješenje je korištenje pomoćne strukture, kao što je rječnik, u kojoj bi se svakom elementu u hrpi pridružio čvor hrpe u kojoj je element spremljen. Rječnik bi bilo potrebno ažurirati prilikom izbacivanja elemenata iz hrpe.

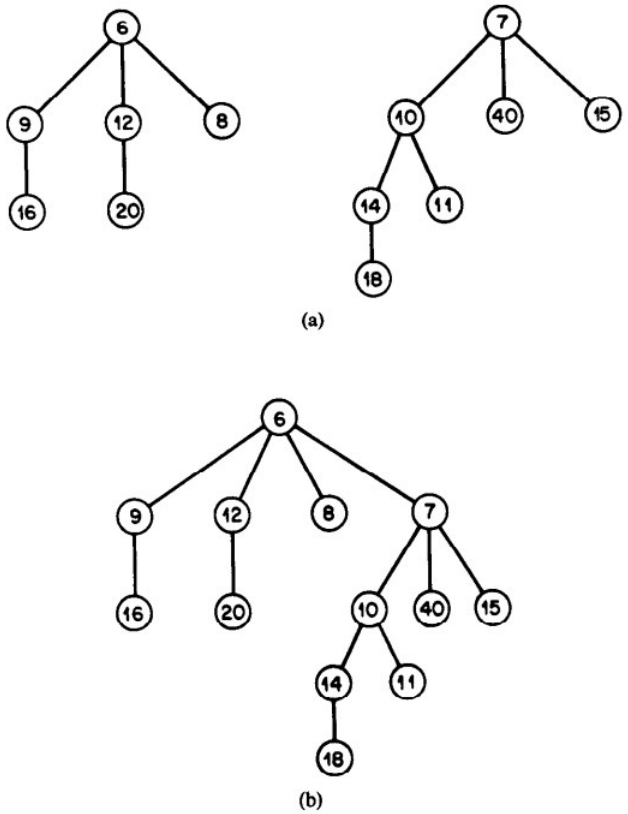
Postoje još dvije operacije nad čvorovima koje služe za jednostavnije izvršavanje osnovnih operacija. To su:

- *link(x,y)* – neka su x i y korijeni stabala u F-hrpi koje treba spojiti u jedno stablo. Ako je $x.key < y.key$, y postaje dijete čvora x. Ukoliko vrijedi $x.key \geq y.key$, x postaje dijete čvora y. Nakon spajanja je potrebno ažurirati rang i listu djece čvora s manjim ključem.
- *cut(x)* – ako je x korijen nekog stabla, ova operacija ne radi ništa. Ako x ima roditelja, x se uklanja iz liste djece njegovog roditelja i dodaje u listu korijena. Ovom operacijom rang roditelja čvora x smanjuje se za 1.

Obje navedene operacije imaju vremensku složenost $O(1)$ zbog korištenja dvostruko vezanih cirkularnih lista u prikazu F-hrpe. Primjer rada operacije *link()* dan je na slici 2.2.

Za kreiranje nove hrpe operacijom *makeHeap()* dovoljno je vratiti *null* pokazivač koji predstavlja praznu F-hrpu. Operacija *insert(x)* stvara novo stablo koje se sastoji samo od x i dodaje ga u F-hrpu, tj. dodaje čvor s elementom x u listu korijena F-hrpe. Operacija *findMin()* dohvaća minimalni čvor i vraća element spremljen u njemu. Očito je da navedene operacije ima vremensku složenost $O(1)$.

Spajanje dviju F-hrpa operacijom *meld(H_1, H_2)* se ostvaruje konkatenacijom lista korijena od H_1 i H_2. Potrebno je još odrediti minimalni čvor nove hrpe koji se definira kao manji od minimalnih čvorova H_1 i H_2. Ovako definirano spajanje dviju hrpa se izvodi u $O(1)$. S obzirom da je podržano efikasno spajanje dviju F-hrpa, Fibonaccijeve hrpe se ubrajaju u grupu hrpa zvanu *spojive hrpe* (engl. meldable heaps).



Slika 2.2: Primjer rada operacije *link()*, pri čemu su elementi stabala reprezentirani svojim ključevima. a) Stabla s korijenima 6 i 7 prije spajanja, b) Stablo nakon spajanja.

Operacija deleteMin

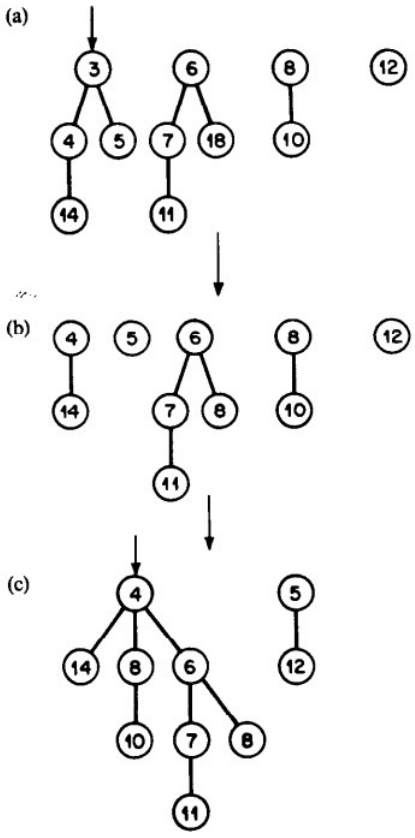
Operacija *deleteMin()* je vremenski najzahtjevnija operacija na F-hrpi. Sama operacija se odvija u 3 koraka:

1. Djeca najmanjeg čvora x F-hrpe se dodaju u listu korijena, element spremljen u x se sprema i čvor x se briše iz liste korijena.

2. Traže se dva stabla s korijenima istog ranga i pomoću operacije *link()* se spajaju u jedno stablo, pri čemu korijen novog stabla ima rang za jedan veći od korijena starih stabala. Ovaj korak se ponavlja sve dok svi korijeni stabala nemaju različite rangove.

3. Iterira se kroz novu listu korijena i traži element s najmanjim ključem kako bi se odredio novi minimalni čvor. Vrijednost starog minimalnog elementa se vraća kao rezultat operacije.

Drugi korak ove operacije se naziva konsolidacija liste korijena. Kako bi se taj korak efikasno izveo, potrebno je pomoćno polje indeksirano rangovima korijena, od 0 do maksimalnog mogućeg ranga. Svako mjesto u polju sadrži pokazivač na neki korijen stabla ili *null* pokazivač. Nakon što se izbriše najmanji element, iterira se po listi korijena i pokazivač na trenutni korijen se sprema u listu na mjesto određeno rangom tog korijena. Ukoliko je mjesto u polju već zauzeto, zna se da trenutni i spremljeni korijen imaju jednake rangove pa ih se spaja pomoću *link()* u novo stablo, nakon čega se korijen novonastalog stabla pokuša spremiti u sljedeću poziciju u polju. Drugi korak operacije završava kada se svi korijeni spreme u polje. Ukupno vrijeme *deleteMin()* operacije je jednako zbroju maksimalnog ranga novonastalih korijena i broja spajanja stabala. Primjer ove operacije dan je na slici 2.3.



Slika 2.3: Primjer rada operacije deleteMin(). a) Hrpa prije operacije, b) Hrpa nakon izbacivanja minimalnog elementa s ključem 3, c) Hrpa nakon spajanja redom korijena 4 i 8, 4 i 6, 5 i 12.

Operacije decreaseKey i delete

Da bi se obavila operacija *decreaseKey(k,i,H)*, pronađe se čvor x koji sadrži i, te se ključu od i smanji vrijednost za nenegativnu vrijednost k. Ovom akcijom se svojstvo hrpe u podstablu generiranom s x neće narušiti, no x može postati manji od svog roditelja. Kako bi se uspostavilo svojstvo hrpe u stablu u kojem se x nalazi, x se pozivom operacije *cut(x)* briše iz liste djece svog roditelja i podstablo generirano s x postaje novo stablo hrpe H. Vrijeme izvršavanja ove operacije je $O(1)$.

Operacija *delete(i,H)* je slična prethodno opisanoj operaciji. Kao i u prethodnoj operaciji, pronađe se čvor x koji sadrži element i, te se x izbaci iz liste djece roditelja operacijom *cut()*. Cilj je izbaciti x iz hrpe, stoga se on ne dodaje u listu korijena, već njegova djeca, nakon čega se x briše. Ako je x korijen stabla, x nema roditelja, pa se samo ukloni iz liste korijena. Ako je x minimalni čvor hrpe, postupa se kao u operaciji *deleteMin()*. Vrijeme potrebno za izvršavanje opisane operacije je $O(1)$, ako i nije minimalni element.

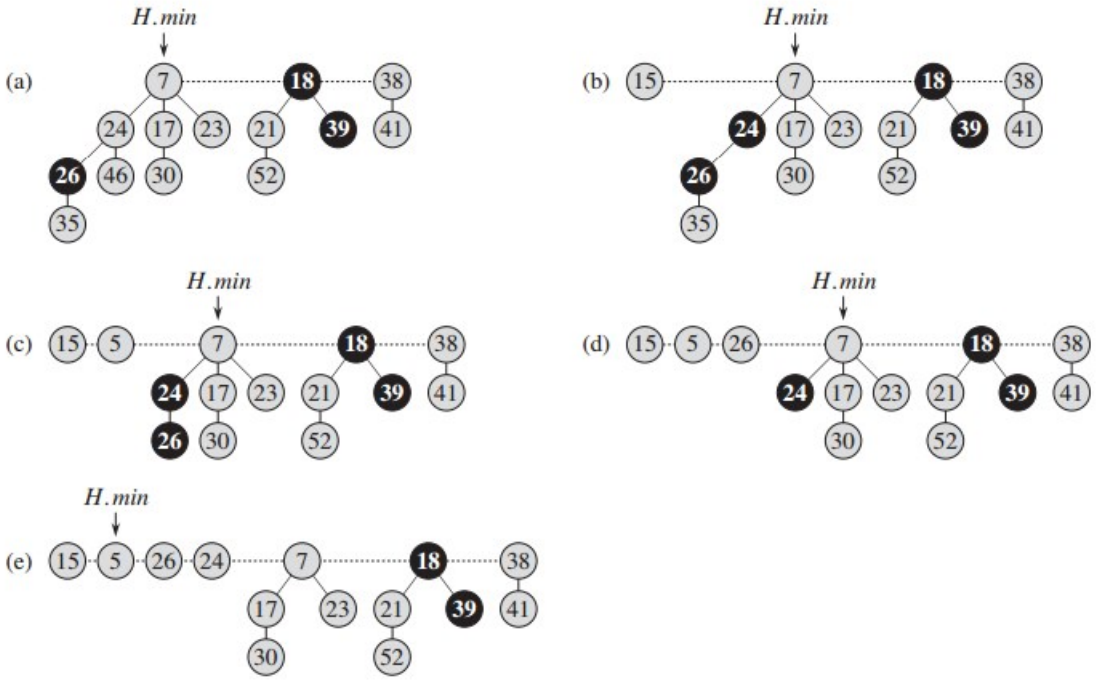
Postoji još jedan detalj implementacije Fibonaccijeve hrpe, koji je bitan za vremensku složenost operacija na hrpi. Nakon što korijen x postane dijete nekog čvora operacijom *link()* i izgubi dvoje svoje djece operacijom *cut()*, x se odvaja od svog roditelja s *cut(x)* i dodaje u listu korijena. Ovakav rez se naziva kaskadni rez (engl. cascading cut). Jedan poziv operacija *delete()* i *decreaseKey()* može uzrokovati veliki broj kaskadnih rezova.

Uloga *marked* parametra, koji je ranije bio naveden, je pamćenje broja djece koje je čvor izgubio. Nakon što čvor x postane dijete u procesu spajanja stabala, x se definira kao neoznačen. Ako je x neoznačen kada izgubi jedno svoje dijete, *x.marked* se postavi na *true*. Ukoliko je x već bio označen, na njemu se obavi kaskadni rez. Korijen stabla je uvijek neoznačen i na njemu nema smisla obaviti kaskadni rez. Primjer kaskadnog reza i operacije *decreaseKey()* prikazan je na slici 2.4.

2.3 Analiza složenosti operacija

Amortizacijska analiza složenosti

Amortizacijska analiza složenosti se javlja kao alternativa vjerojatnosnoj analizi složenosti operacija na strukturama podataka [9]. Nad strukturama podataka često se obavlja niz operacija, pa je poželjno promatrati vrijeme izvođenja cijelog niza operacija, umjesto svake operacije zasebno. Ukoliko se promatra svaka operacija zasebno, analiza složenosti u najgorem slučaju može biti prepesimistična, jer neke operacije mogu imati učinak na vremena izvršavanja drugih operacija. Promatranje prosječnog vremena izvršavanje zasebne operacije ovisi o pravilnoj vjerojatnosnoj analizi koja se u nekim slučajevima može pokazati pogrešnom.



Slika 2.4: Primjer kaskadnog reza pri čemu su crnom bojom prezentirani označeni čvorovi a) Početna hrpa b) Hrpa nakon smanjenja ključa 46 na 15, c) Hrpa nakon smanjenja ključa 35 na 5 d) Hrpa nakon kaskadnog reza elementa s ključem 26, e) Hrpa nakon kaskadnog reza elementa s ključem 24.

U amortizacijskoj analizi, vrijeme potrebno da bi se obavio niz operacija nad strukturom se razdijeli na sve obavljenje operacije. Amortizacijskom analizom se može pokazati da je prosječni "trošak" operacije u nizu operacija mali, iako jedan poziv operacije može biti "skup".

Postoji više tehnika koje se koriste u amortizacijskoj analizi složenosti, no za potrebe analize operacija Fibonaccijeve hrpe koristiti će se *metoda potencijala*. Metoda funkcionira na sljedeći način. Neka je D_0 početna struktura podataka na kojoj se obavi s operacija. Za svaki $i = 1, \ldots, s$, neka je c_i stvarni trošak i-te operacije i neka je D_i struktura koja nastane nakon izvođenja i-te operacije na strukturi D_{i-1}. *Funkcija potencijala* Φ preslikava svaku strukturu D_i u realni broj $\Phi(D_i)$, koji se naziva *potencijal* od D_i. Amortizirani trošak i-te operacije $\hat{c}_i$ je dan s:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{2.1}$$

Amortizirani trošak je zbroj stvarnog troška operacije i rasta potencijala zbog same opera-

cije. Ukupni amortizirani trošak s operacija je:

$$\sum_i \hat{c}_i = \sum_i c_i + \Phi(D_s) - \Phi(D_0). \tag{2.2}$$

Ukoliko se pokaže da vrijedi $\Phi(D_s) \geq \Phi(D_0)$, može se amortiziranim troškom ograničiti stvarni utrošak vremena. U praksi se unaprijed ne zna koliko će operacija nad strukturom biti pozvano, pa je potebno da nakon svake operacije i vrijedi $\Phi(D_i) \geq \Phi(D_0)$. Često je korisno definirati $\Phi(D_0) = 0$ i pokazati da za svaki i vrijedi $(D_i) \geq 0$.

Svojstva Fibonaccijeve hrpe

Fibonaccijeva hrpa ima dva svojstva koja su nužna za analizu složenosti operacija:

1. Veličina svakog stabla je barem eksponencijalna u odnosu na rang korijena stabla.
2. Broj kaskadnih rezova koji se mogu dogoditi tijekom niza operacija na F-hrpi je ograničen brojem operacija *decreaseKey()* i *delete()*.

Kaskadni rezovi na hrpi su nužni kako bi prvo svojstvo bilo očuvano, dok pravilo rezanja čvora nakon drugog izgubljenog djeteta služi kako bi se ograničio broj kaskadnih rezova. Za amortizacijsku analizu operacija na Fibonaccijevoj hrpi, potrebno je pokazati da je gornja granica $D(n)$ ranga proizvoljnog čvora u hrpi s n elemenata jednaka $O(n)$.

Lema 2.3.1. *Neka je x proizvoljni čvor Fibonaccijeve hrpe i pretpostavimo da je $r(x) = k$, pri čemu $r(x)$ označava rang čvora x. Neka su s $y_1, y_2, .., y_k$ označena djeca od x u poretku u kojem su spojena s x, od najranijeg do zadnjega. Tada je $r(y_1) \geq 0$ i $r(y_i) \geq i-2$, za svaki $i = 2, 3, \ldots, k$.*

Dokaz. Očito vrijedi $r(y_1) \geq 0$. Za $i \geq 2$, primijetimo da su $y_1, y_2, \ldots, y_{i-1}$ bili djeca od x kada je y_i povezan s x, pa je moralo vrijediti $r(y_i) = i - 1$, jer se y_i mogao povezati sa x ako je vrijedilo $r(x) = r(y_i)$. Otada je čvor y_i mogao izgubiti najviše jedno dijete, jer bi u protivnom bio odrezan od x. Stoga vrijedi $r(y_i) \geq i - 2$. □

Definicija 2.3.2. *Neka je $k \geq 0$. S F_k označava se k-ti Fibonaccijev broj definiran s:*

$$F_k = \begin{cases} 0, & k = 0 \\ 1, & k = 1 \\ F_{k-1} + F_{k-2}, & k \geq 2. \end{cases}$$

Sljedeće dvije leme daju neka bitna svojstva potrebna za lemu kojom će biti objašnjeno ime Fibonaccijeve hrpe.

Lema 2.3.3. *Za svaki $k \geq 0$ vrijedi*

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Dokaz. Dokaz se provodi trivijalno indukcijom [8]. □

Lema 2.3.4. *Za svaki $k \geq 0$ vrijedi*

$$F_{k+2} \geq \phi^k,$$

pri čemu se ϕ zove zlatni rez i dan je s $\phi = (1 + \sqrt{5})/2$.

Dokaz. Dokaz se provodi indukcijom po k [8]. □

Korolar 2.3.5. *Neka je x proizvoljni čvor u Fibonaccijevoj hrpi i neka je $k = r(x)$. Označimo sa $size(x)$ broj potomaka od x, uključujući i njega. Tada je $size(x) \geq F_{k+2} \geq \phi^k$.*

Dokaz. Neka je S_k minimalni mogući broj potomaka čvora ranga k. Očito je $S_0 = 1$ i $S_1 = 2$. Iz leme 2.3.1 slijedi da je $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ za $k \geq 2$. Indukcijom po k i koristeći leme 2.3.3, 2.3.4 dobije se $size(x) \geq S_k \geq F_{k+2} \geq \phi^k$. □

Fredman i Tarjan navode da je upravo ovaj korolar izvor imena opisane strukture podataka.

Ostalo je još pokazati da je rang čvora u hrpi s n elemenata ograničen s $O(\log n)$. Taj rezultat daje sljedeći korolar.

Korolar 2.3.6. *Maksimalni rang $D(n)$ proizvoljnog čvora Fibonaccijeve hrpe s n elemenata je ograničen s $\lfloor \log_\phi n \rfloor$.*

Dokaz. Neka je x neki čvor Fibonaccijeve hrpe s n elemenata i neka je k njegov rang. Po prethodnom korolaru znamo da vrijedi $n \geq size(x) \geq \phi^k$. Logaritmiranjem ovog izraza po bazi ϕ dobije se $k \leq \log_\phi n$. □

Amortizirano vrijeme operacija na Fibonaccijevoj hrpi

Na Fibonaccijevoj hrpi H funkcija potencijala definira se s:

$$\Phi(H) = t(H) + 2m(H), \tag{2.3}$$

pri čemu je s $t(H)$ broj stabala u hrpi H, a $m(H)$ broj označenih čvorova u hrpi H. Prazna hrpa ima potencijal 0 jer su $t(H)$ i $m(H)$ jednaki 0 i potencijal je očito nenegativan za sve naredne operacije. Zbog toga je moguće s ukupnim amortiziranim vremenom izvršavanja operacija ograničiti stvarno vrijeme izvršavanja operacija.

Operacije *makeHeap()*, *findMin()*, *insert()* i *merge()* imaju amortizirano vrijeme izvođenja $O(1)$. Naime, potrebno im je $O(1)$ vremena za obavljanje svoje zadaće i samo *insert()* povećava potencijal za 1, dok ostale operacije ne mijenjaju potencijal.

Kako bi se izračunalo amortizirano vrijeme operacije *deleteMin()* potrebno je izračunati njezin stvarni trošak vremena i promjenu potencijala. Spajanje djece minimalnog čvora s listom korijena se obavlja u $O(1)$ vremena. Kao što je već pokazano, čvor može imati najviše $D(n) = O(\log n)$, pa je nakon ovog koraka lista korijena veličine $D(n) + t(H) - 1$. Operacija spajanja stabala, koja je izvediva u vremenu $O(1)$, može biti izvršena maksimalno $D(n) + t(H) - 1$ puta. Ukupan stvarni trošak ove operacije tada je $O(D(n) + T(h))$. Potencijal na početku operacije brisanja je $t(H) + 2m(H)$, a potencijal na kraju operacije je $(D(n) + 1) + 2m(H)$, jer najviše $D(n) + 1$ korijena ostane i nijedan čvor se ne označi tijekom brisanja. Uzimajući ovo u obzir, amortizirano vrijeme izvođenja operacije *deleteMin()* je najviše

$$\begin{aligned} &O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &\quad = O(D(n)) + O(t(H)) - t(H) = O(D(n)) = O(\log n), \end{aligned}$$

jer je moguće skalirati jedinice potencijala kako bi se dominirala skrivena konstanta u članu $O(t(H))$. Moglo bi se reći da je trošak operacija *link()* "plaćen" smanjenjem potencijala zbog manjeg broja stabala na kraju operacije.

Smanjivanje ključa u operaciji *decreaseKey()* i odvajanje od roditelja se obavlja u vremenu $O(1)$. Pretpostavimo da ova operacija uzrokuje c kaskadnih rezova, pri čemu je za jedan kaskadni rez potrebno vrijeme $O(1)$, pa je ukupno vrijeme izvršavanja operacije $O(c)$. Broj stabala nakon operacije je jednak $T(H) + c$, dok je maksimalni broj označenih čvorova $m(H) - c + 2$ ($c - 1$ čvor više nije označen i zadnji kaskadni rez je možda označio čvor). Slijedi da je razlika potencijala tada najviše

$$((t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H)) = 4 - c.$$

Iz ovoga slijedi da je amortizirana složenost operacije *decreaseKey()* najviše

$$O(c) + 4 - c = O(1),$$

pri čemu se skaliraju jedinice potencijala kako bi se dominirala skrivena konstanta u članu $O(c)$.

Kombinacijom analiza za operacije *deleteMin()* i *decreaseKey()* se pokazuje da je amortizirana složenost operacije *delete()* jednaka $O(\log n)$. Naime, *delete()* se ponaša slično kao i *decreaseKey()*, no umjesto čvora koji se briše, u listu korijena se dodaju njegova djeca čime broj stabala, pa tako i potencijal, može narasti za najviše $O(\log n)$.

2.4 Varijante Fibonaccijeve hrpe

Fredman i Tarjan predložili su još neke varijante Fibonaccijeve hrpe, koje su pogodnije ukoliko se zahtijevaju dodatne operacije. Također, tijekom godina znanstvenici su pokušavali dobiti strukture za implementaciju prioritetnog reda s boljim asimptotskim vremenima izvršavanja, kao što su Brodalov red i stroga Fibonnacijeva hrpa. Slijedi kratki opis navedenih struktura.

Fibonaccijeve hrpe s praznim čvorovima

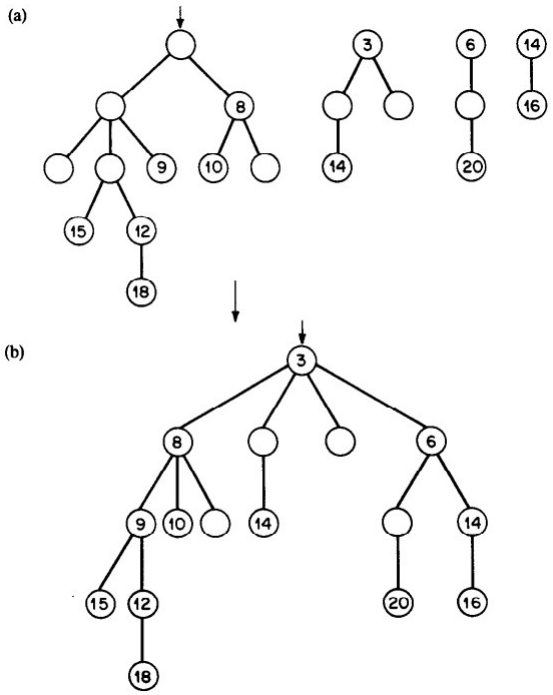
Za brisanje proizvoljnog elementa u F-hrpi, amortizirana ocjena $O(\log n)$ može, u nekim slučajevima, biti precijenjena. Primjerice, želi li se obrisati cijela F-hrpa, dovoljno je proći po svim stablima u hrpi i obrisati jedan po jedan čvor. Ovo razmatranje može se generalizirati "lijenim" brisanjem elemenata koje je predstavljeno praznim čvorovima u Fibonaccijevoj hrpi.

Prilikom obavljanja operacija *delete()* i *deleteMin()*, umjesto brisanja čvora s danim elementom, izbriše se element iz čvora, a čvor se označi praznim. Ovako definirano brisanje elemenata je moguće obaviti u vremenu $O(1)$. S obzirom da ove operacije mogu izbrisati minimalni element i ostaviti minimalni čvor praznim, potrebno je modificirati *meld()* i *findMin()* operacije. Ako se spajaju dvije hrpe, od kojih jedna ima prazan minimalni čvor, minimalni čvor nove hrpe postaje prazni minimalni čvor stare hrpe. Za pronalazak najmanjeg elementa u hrpi, ako je minimalni čvor prazan, potrebno je proći kroz svako stablo i uništiti sve prazne čvorove od korijena do prvog nepraznog čvora. Nakon toga se konsolidira lista korijena kao i u operaciji *deleteMin()* standardne F-hrpe, što daje garanciju da su svi korijeni neprazni. Primjer ove operacije je dan na slici 2.5. Može se pokazati da ovako definirana *findMin()* operacija ima amortiziranu vremensku složenost $O(l(\log(n/l) + 1))$, pri čemu je l broj uništenih praznih čvorova, a n broj elemenata hrpe.

Opisano "lijeno" brisanje elemanata je korisno za aplikacije u kojima je moguće implicitno identificirati obrisane elemente. Glavna mana ovakvog načina brisanja je što može biti memorijski neučinkovito ako se elementi često brišu i dodaju u hrpu.

Fibonaccijeve hrpe s dobrim i lošim stablima

Umjesto korištenja praznih čvorova moguće je podijeliti stabla u F-hrpi na *dobra* i *loša* stabla. Minimalni čvor hrpe se definira kao korijen dobrog stabla s minimalnim ključem. Prilikom ubacivanja novog elementa, novonastalo stablo s jednim čvorom se označava dobrim. Spajanjem dviju hrpa, spajaju se skupovi dobrih i skupovi loših stabala hrpa. Sva stabla nastala kaskadnim rezovima nakon operacija *decreaseKey()* označavaju se kao dobra stabla. Operacija *delete()* se obavlja kao i kod standardne F-hrpe, osim u slučaju brisanja minimalnog čvora. U tom slučaju sva podstabla, kojima su korijeni djeca minimalnog



Slika 2.5: Primjer rada operacije findMin() na hrpi s praznim čvorovima. a) Hrpa prije operacije. b) Hrpa nakon operacije, podsatbla s korijenima 15,12,9 i 8 postaju stabla, nakon čega se konsolidiraju korijeni.

čvora, postaju loša stabla. Konsolidacija liste korijena se obavlja tek nakon poziva operacije *findMin()* u slučaju postojanja loših stabala. Tada se povežu sva stabla istog ranga i sva stabla označe dobrima, nakon čega se pronađe novi minimalni element i vrati kao rezultat operacije. U ovoj varijanti Fibonaccijeve hrpe, operacije *makeHeap()*, *insert()*, *meld()* i *decreaseKey()* imaju amortiziranu složenost $O(1)$. Amortizirana složenost operacije *findMin()* je $O(\log(n/l) + 1)$, pri čemu je l broj operacija brisanja elementa između dva poziva operacije *findMin()*. Prednost ove varijante, u odnosu na prethodnu, je što ne zauzima memoriju s izbrisanim čvorovima.

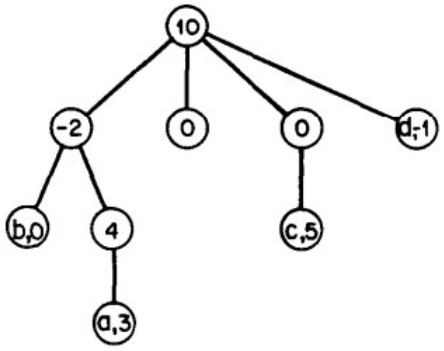
Fibonaccijeve hrpe s implicitnim ključevima

F-hrpa s implicitnim ključevima je dizajnirana u cilju efikasnog podržavanja sljedeće operacije:

increaseAllKeys(k,h) – povećava ključeve svih elemenata hrpe h za realnu vrijednost k.

Kako bi se implementirala ova operacija, ključevi se ne reprezentiraju unutar hrpe, već

korištenjem pomoćne strukture podataka. Jedna od prikladnih struktura za spremanje ključeva je kompresirano stablo, pri čemu je svakoj hrpi pridruženo jedno kompresirano stablo. Čvorovi u stablu sadržavaju vrijednost i moguće neki element. Čvorovi koji sadržavaju element mogu biti samo listovi u stablu. Ključ elementa *i* spremljenog u čvoru *x* se računa zbrajanjem vrijednosti spremljenih u čvorove koji se nalaze na putu od *x* do korijena stabla (uključujući i sam *x*). Primjer ovakvog prikaza dan je na slici 2.6. Operacija *increaseAllKeys(k,h)* se trivijalno implementira uvećavanjem ključa spremljenog u korijenu za *k*.



Slika 2.6: Primjer jednog kompresiranog stabla. Elementi spremljeni u stablu su označeni s *a*, *b*, *c* i *d*. Ključ elementa *a* je jednak $3 + 4 - 2 + 10 = 15$.

Pokazuje se da je u većini aplikacija, koje zahtijevaju operaciju *increaseAllKeys()*, vrijeme manipuliranja kompresiranim stablom dominirano vremenom manipuliranja hrpom. Kompresirano stablo je u potpunosti odvojeno od hrpe, pa ga je moguće koristiti s bilo kojom varijantom hrpe.

Brodalov red

Brodal je 1996. godine u [2] predstavio prvu varijantu hrpe koja postiže jednaka asimptotska vremena operacija na hrpi kao i Fibonaccijeva hrpa ($O(1)$ za *findMin()*, *insert()* i *decreaseKey*, $O(\log n)$ za *deleteMin()* i *delete()*). Hrpa se po autoru naziva Brodalov red. Za razliku od Fibonaccijeve hrpe, sva vremena operacija na Brodalovom redu su dana u najgorem slučaju. Brodalov red zahtijeva korištenje polja i RAM memorije za uspješnu implementaciju. Hrpa je veoma kompliciranа za implementaciju, pa ima više teorijsku nego praktičnu važnost.

Stroga Fibonaccijeva hrpa

Brodal, Lagogiannis i Tarjan su 2012. godine u [3] predstavili strogu Fibonaccijevu hrpu kojom poboljšavaju Brodalov red. Stroga Fibonaccijeva hrpa ima jednaka vremena izvršavanja operacija kao i Brodalov red, no za razliku od Brodalovog reda temeljena je na pokazivačima, pa nužno ne zahtijeva RAM model za implementaciju. Stroga Fibonaccijeva hrpa ima strukturu stabla na kojem vrijedi svojstvo hrpe i zasniva se na činjenici da su rangovi čvorova logaritamski ograničeni brojem čvorova u hrpi. Stroga Fibonaccijeva hrpa, jednako kao i Brodalov red, zbog svoje kompliciranosti nije pronašla svoje mjesto u praksi.

Poglavlje 3

Primjena Fibonaccijeve hrpe

Fibonaccijeva hrpa je primarno smišljena kako bi se poboljšalo asimptotsko vrijeme izvršavanja algoritama na grafovima. Dva su bitna algoritma uspješno teorijski ubrzana korištenjem Fibonaccijeve hrpe kao implementacije prioritetnog reda: Dijkstrin algoritam najkraćeg puta i Primov algoritam.

Definirajmo prvo neke pojmove koji će biti potrebni u opisu spomenutih algoritama.

Definicija 3.0.1. ***Graf*** *je uređeni par $G = (V, E)$, gdje je $\emptyset \neq V = V(G)$ skup vrhova (engl. vertex), $E = E(G)$ skup bridova (engl. edge), a svaki brid $e \in E$ spaja dva vrha $u, v \in V$ koji se zovu krajevi od e. Svakom bridu $e = (u, v) \in E$ može biti pridružen realni broj $l(e)$ (ili $l(u, v)$) koji se naziva težina ili duljina brida e.*

Definicija 3.0.2. ***Put*** *od vrha u do vrha v je niz vrhova $u, x_1, ..., x_n, v$ pri čemu su svaka dva susjedna vrha u nizu povezana bridom. Duljina puta je zbroj duljina bridova na putu od u do v. Najkraći put od u do v je put od u do v s najmanjim zbrojem bridova.*

Definicija 3.0.3. *Za podgraf $T = (V', E')$ grafa $G = (V, E)$ kažemo da je stablo ako ne postoji ciklus u T. Za stablo T kažemo da je razapinjuće stablo grafa G ako je $V' = V$. Minimalno razapinjuće stablo je razapinjuće stablo s najmanjim zbrojem duljina bridova.*

3.1 Dijkstrin algoritam najkraćeg puta

Neka je G usmjereni graf sa svim bridovima nenegativne duljine. Označimo sa n broj vrhova u G i s m broj bridova u G. U grafu odaberimo jedan istaknuti čvor s koji se naziva početni vrh. Pretpostavimo da postoji put od s do svakog vrha $v \in G$, pa vrijedi $m \geq n - 1$. Problem najkraćeg puta je tada problem pronalaska najkraćeg puta od s do svakog vrha v grafa G.

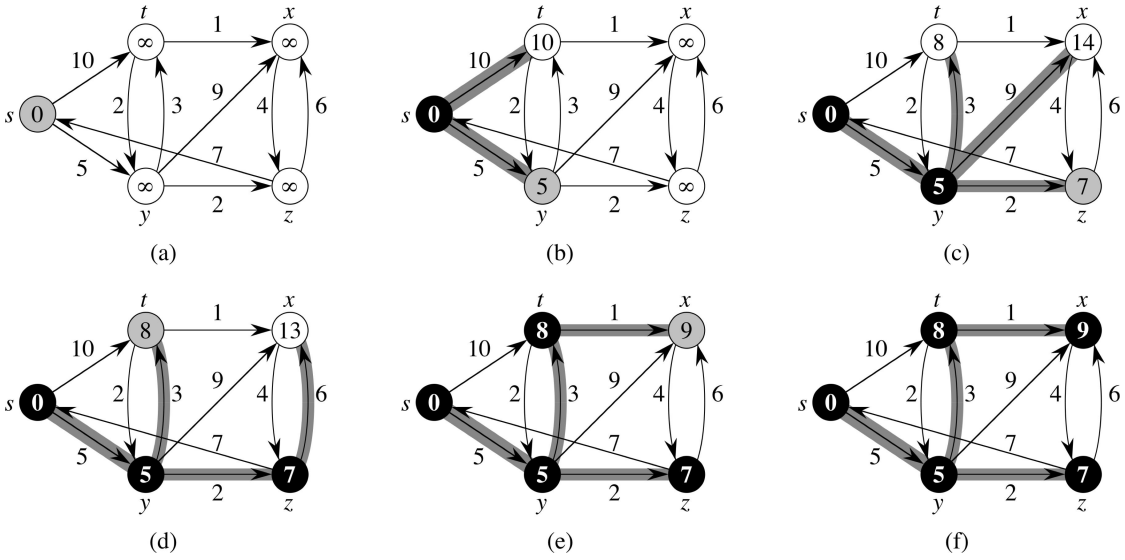
Dijkstrin algoritam [4] za rješavanje problema najkraćeg puta koristi *funkciju udaljenosti d* koja preslikava vrhove u realne brojeve i ima sljedeća svojstva:

1. Za svaki vrh v za takav da je $d(v)$ konačan, postoji put od s do v duljine $d(v)$.
2. Kada algoritam završi, $d(v)$ je duljina najkraćeg puta od s do v.

Svaki vrh tijekom rada algoritma može biti u jednom od tri stanja: *neoznačen*, *označen* ili *pregledan*. Algoritam se sastoji od sljedeća tri koraka:

1. Postavi $d(s) = 0$, $d(v) = \infty$ za svaki $v \neq s$ i promijeni stanje vrha s u označen.
2. Odaberi vrh v iz skupa označenih vrhova za kojeg je $d(v)$ minimalna i označi ga pregledanim. Za svaki brid (v, w) takav da je $d(v) + l(v, w) < d(w)$ postavi $d(w) = d(v) + l(v, w)$ i promijeni stanje vrha w u označen.
3. Ukoliko ima vrhova sa stanjem označen, ponovi 2. korak. U suprotnom, algoritam završava.

Uvjet nenegativnosti duljina bridova u grafu je nužan za korektan rad algoritma jer time postoji garancija da vrhovi koji su pregledani ne mogu ponovno postati označeni. Primjer rada Dijsktrinog algoritma je dan na slici 3.1.



Slika 3.1: Primjer rada Dijkstrinog algoritma. Bijelom bojom su označeni vrhovi u stanju neoznačen, sivom označeni vrhovi, a crnom pregledani vrhovi. Unutar vrhova je upisana vrijednost funkcije d za taj vrh.

Za efikasan rad Dijkstrinog algoritma potreban je brz način pronalaska minimalnog označenog vrha u drugom koraku algoritma. Fibonaccijeva hrpa, sa svojim podržanim operacijama, je idealan kandidat za spremanje označenih vrhova kojima se kao ključ pridruži

vrijednost funkcije d. Prvi korak algoritma zahtijeva po jedan poziv *makeHeap()* i *insert()* operacije. Svaki korak pregledavanja vrha zahtijeva jedan poziv *deleteMin()* metode. Za svaki brid pregledanog vrha, takav da vrijedi $d(v) + l(v, w) < d(w)$, potrebna je *insert()* ($d(w) = \infty$) ili *decreaseKey()* ($d(w) < \infty$) operacija. Iz ovoga slijedi da je u cijelom algoritmu potreban jedan poziv *makeHeap()* operacije, n poziva *insert()* i *deleteMin()* operacija, te najviše m *decreaseKey()* operacija. Ukupno vrijeme potrebno za operacije na F-hrpi je $O(n \log n + m)$, dok ostali zadaci u Dijkstrinom algoritmu zahtijevaju $O(n + m)$ vremena. Vrijeme izvođenja Dijkstrinog algoritma korištenjem F-hrpe je stoga $O(n \log n + m)$, što je bolje od $O(m \log_{m/n+2} n)$, najboljeg poznatog vremena za Dijkstrin algoritam prije pojave Fibonaccijeve hrpe. Pokazalo se da se ovo vrijeme može unaprijediti na $O(m \log \log C)$, ako su duljine bridova cijeli brojevi ograničeni konstantnom C [1].

Često je potrebno, uz duljinu najkraćeg puta, odrediti i najkraći put, tj. vrhove na najkraćem putu. Za svaki vrh v s $p(v)$ označimo prethodnika vrha v na najkraćem putu od s do v. Prilikom zamjene vrijednosti $d(w)$ s $d(v) + l(v, w)$ postavi se $p(w) = v$. Najkraći put se na kraju algoritma može dobiti praćenjem prethodnika od v prema s. Ovakva modifikacija algoritma ne mijenja ukupno vrijeme izvođenja jer je potrebno samo $O(m)$ dodatnih koraka.

Poboljšanjem vremena izvršavanja Dijkstrinog algoritma se, također, poboljšavaju vremena sljedećih algoritama u kojima se Dijkstrin algoritam koristi kao potprogram:

- Problem najkraćeg puta svih parova vrhova u grafu je moguće riješiti s n iteracija Dijkstrinog algoritma. Ukupno vrijeme rješavanja ovog problema je tada $O(n^2 \log n + nm)$.

- Problem pridruživanja se, također, može riješiti u vremenu $O(nm)$ uz n iteracija Dijkstrinog algoritma, pa je ukupno vrijeme cjelokupnog algoritma, uz korištenje Fibonaccijeve hrpe, jednako $O(n^2 \log n + nm)$.

3.2 Primov algoritam

Problem najmanjeg razapinjućeg stabla grafa G je jedan od poznatijih problema kombinatorne optimizacije. Potrebno je, za dani neusmjereni graf G, odrediti njegovo najmanje razapinjuće stablo. Određivanje najmanjeg razapinjućeg stabla moguće je napraviti korištenjem pohlepnog pristupa, koji daje jednostavan i korektan algoritam. Jedan od takvih pohlepnih algoritama je Primov algoritam. Algoritam se pripisuje neovisno Primu i Dijkstri, iako ga je 1930. godine prvi otkrio Jarnik [7]. Primov algoritam radi na sljedeći način:

1. Odabire se proizvoljni vrh v u grafu G. Kreira se stablo T s v kao njegovim jedinim elementom.

2. Odabire se brid minimalne duljine takav da je točno jedan njegov kraj u T. Stablo T se proširuje dodavanjem odabranog brida u T.

3. Prethodni korak se ponavlja $n - 1$ puta, pri čemu je n broj vrhova u G. Nakon $n - 1$ koraka, stablo T će biti minimalno razapinjuće stablo od G.

Očito je da je, za korektan rad ovog algoritma, potrebno da je G povezan. Primov algoritam, za razliku od Dijkstrinog, ne postavlja uvjete na duljine bridova, tj. algoritam radi korektno za proizvoljne duljine bridova u grafu.

Primov algoritam je sličan Dijkstrinom algoritmu, pa se može implementirati na sličan način korištenjem Fibonaccijeve hrpe. Svaki vrh v, koji nije u stablu T, se sprema u Fibonaccijevu hrpu, pri čemu je ključ $d(v)$ od v dan kao duljina najkraćeg brida koji spaja v sa stablom T. Uz duljinu brida, pamti se i brid koji ima najmanju duljinu kako bi se na kraju algoritma moglo rekonstruirati stablo T. Na početku algoritma se definira $d(s) = 0$ i $d(v) = \infty$ za svaki $v \neq s$. Korak odabira najmanjeg brida se implementira na sljedeći način:

Korak spajanja: Odabire se vrh v s minimalnim konačnim ključem $d(v)$ i postavlja se $d(v) = -\infty$. Za svaki brid (v, w) takav da je $l(v, w) < d(w)$, $d(w)$ se zamjenuje s $l(v, w)$ i definira se $e(w) = (v, w)$.

Opisani korak se ponavlja dok ključevi svih vrhova ne postanu jednaki $-\infty$. Svrha korištenja $-\infty$ je označavanje vrhova dodanih u stablo. Minimalno razapinjuće stablo je na kraju algoritma definirano s $\{e(v) : v \in G, v \neq s\}$.

Iskoristi li se Fibonaccijeva hrpa za spremanje vrhova s konačnim ključevima, rad algoritma zahtijeva n operacija *deleteMin()* i $O(m)$ operacija *insert()* ili *decreaseKey()*. Ukupno vrijeme izvršavanja Primovog algoritma je tada dano s $O(n \log n + m)$, što je jednako vremenu izvršavanja Dijkstrinog algoritma. Najbolja poznata ograda za Primov algoritam, prije pojave Fibonaccijeve hrpe, bila je $O(m \log \log_{m/n+2} n)$.

Fredman i Tarjan predložili su modifikaciju Primovog algoritma kojom se može dobiti još bolja ograda za pronalazak minimalnog razapinjućeg stabla. Ideja algoritma je kroz više iteracija izgraditi stablo. Kao i u Primovom algoritmu, stablo se gradi iz jednog vrha, ali sve dok hrpa susjednih vrhova stabla ne dosegne određenu veličinu, nakon čega se stablo počinje graditi, po istom principu, iz nekog drugog vrha. Jedna iteracija algoritma završava kada je svaki vrh u nekom stablu. Prije sljedeće iteracija algoritma potrebno je sabiti svako stablo u takozvani supervrh, tj. svako stablo se u sljedećoj iteraciji promatra kao vrh. Nakon dovoljno iteracija algoritma, ostati će samo jedan supervrh iz kojeg se rekonstruira minimalno razapinjuće stablo.

Svaka iteracija algoritma se sastoji od dva dijela:

Čišćenje bridova. Stabla nastala u prethodnoj iteraciji se nazivaju stara stabla i označavaju se brojem kako bi ih se moglo razlikovati. Svi bridovi između vrhova unutar jednog

starog stabla se brišu. Za svaka dva stara stabla, brišu se svi bridovi koji ih spajaju, osim onoga s najmanjom duljinom. Ovim korakom se obavlja sabijanje stabala u vrh, jer se sada svako stablo može promatrati kao vrh. Proces brisanja bridova je moguć u vremenu $O(m)$ vremenu leksikografskim sortiranjem bridova prema oznaci stabala u kojima se nalaze krajevi bridova. Prolaskom kroz sortirani niz, konstruira se lista incidentnih bridova za svako staro stablo T.

Izgradnja novih stabala: Odabire se početno stablo T_0 i ubacuje u hrpu s ključem $d(T_0) = 0$. Sve dok se hrpa ne isprazni, ili dok hrpa ne naraste na veličinu barem k, dohvaća se stablo T s najmanjim ključem i spaja u stablo koje sadrži T_0, kao i vrhovi u koraku spajanja kod Primovog algoritma. Izgradnja novih stabala završava kada se pregledaju sva stara stabla. Izgradnja novih stabala zahtijeva $O(t \log k + m)$ vremena jer je potrebno najviše t *deleteMin()* operacija na hrpama maksimalne veličine k i $O(m)$ ostalih operacija na Fibonaccijevoj hrpi.

Kako bi se minimiziralo vrijeme potrebno za jednu iteraciju algoritma, potrebno je odabrati optimalnu vrijednost k. Odabere li se za svaku iteraciju algoritma $k = 2^{2m/t}$, pri čemu je m broj bridova grafa i t broj starih stabala na početku iteracija, vrijeme potrebno za jednu iteraciju algoritma je $O(m)$.

Ostalo je još ograničiti broj mogućih iteracija algoritma kako bi se dobilo ukupno vrijeme rada algoritma. Može se pokazati da je broj stabala nakon rada jedne iteracije ograničen odozgo s $2m'/k$, pri čemu je m' broj bridova u grafu na početku iteracije. Ovaj rezultat daje gornju ogradu na broj iteracija algoritma s $\beta(m, n) = \min\{i : \log^{(i)} n \leq m/n\}$, pri čemu je $\log^{(i)} n$ induktivno definiran s $\log^{(0)} n = n$ i $\log^{(i+1)} n = \log \log^{(i)} n$. Iz ovoga slijedi da je složenost ovog algoritma $O(m\beta(m, n))$, što znači da je algoritam polinoman za dovoljno rijetke grafove.

Poglavlje 4

Rezultati i zaključak

Implementacija

Uz ovaj rad priloženo je programsko rješenje u kojem su implementirane različite varijante prioritetnog reda s ciljem testiranja efikasnosti njihovog korištenja u Dijkstrinom i Primovom algoritmu. Programsko rješenje izvedeno je u potpunosti u programskom jeziku Java, uz korištenje standardnih biblioteka.

Prioritetni red je implementiran na 4 načina pomoću: binarne hrpe, Fibonaccijeve hrpe, sortiranog skupa i sortirane vezane liste. Kao sortirani skup je korištena klasa *TreeSet* iz standardne biblioteke koja osigurava sve osnovne operacije u $O(\log n)$ vremenu. Fibonaccijeva hrpa je implementirana na način opisan u drugom dijelu ovog rada. Za sortiranu vezanu listu se zna da nije efikasna implementacija prioritetnog reda, no implementirana je kao jedan primjer implementacije s vremenom izvršavanja $O(n)$. Binarna hrpa je implementirana prema načinu opisanom u prvom dijelu ovog rada. Kako bi se omogućilo $O(1)$ pristupanje čvoru sa spremljenim vrhom v, svaki vrh v sprema pokazivač na čvor u kojem je spremljen v u binarnoj i Fibonaccijevoj hrpi.

Graf je implementiran kao lista vrhova, pri čemu su svi susjedi vrha v spremljeni u listu susjedstva (engl. adjacency list) vrha v. Ovakav pristup manje opterećuje memoriju računala nego korištenje matrice susjedstva (engl. adjacency matrix), posebice kod rijetkih grafova.

Kako bi se omogućilo testiranje rada algoritama na slučajno generiranim grafovima različite gustoće, pri geneniranju grafa dodaje se parametar p. Ovaj parametar određuje gustoću grafa, jer za svaka dva vrha grafa G postoji vjerojatnost p da će postojati brid koji ih povezuje. Težina bridova je generirana kao slučajni cijeli broj između 1 i 1000. U svakom generiranom grafu je osigurana povezanost grafa.

4.1 Rezultati

S obzirom da je cilj rada bio korištenjem Fibonaccijeve hrpe ubrzati rad opisanih algoritama, testirane su brzine izvođenja različitih varijanti algoritama, ovisno o tome koju implementaciju prioritetnog reda koriste. Za svaku unaprijed zadanu veličinu grafa, generirano je 10 grafova na kojima su izvedene sve verzije algoritama. Konačno vrijeme izvođenja je uzeto kao prosjek vremena izvođenja algoritma na svih 10 grafova. Ovakav način je odabran kako bi se smanjila odstupanja pojedinih izvođenja algoritama.

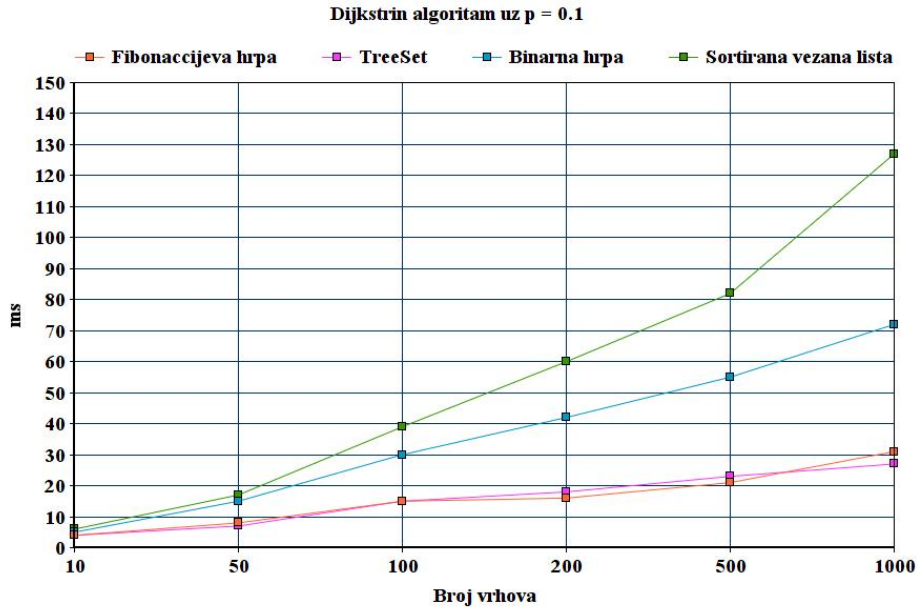
Podaci dobiveni testiranjem raznih verzija Dijkstrinog algoritma prikazani su grafovima 4.1a i 4.1b, iz kojih se odmah može vidjeti neefikasnost implementiranja prioritetnog reda pomoću vezane sortirane liste. Nedostatak ove implementacije se najviše očituje u slučaju gustog grafa, gdje je veći broj poziva operacija *insert()* i *decreaseKey()*. Iako garantira sve operacije u $O(\log n)$ vremenu, *TreeSort* struktura iz standardne biblioteke Jave se također nije pokazala jednako efikasnom kao i binarna hrpa. Binarna hrpa je, usprkos lošijim asimptotskim vremenima operacija od Fibonaccijeve hrpe, uspjela u provedenim testovima pokazati otprilike jednaku uspješnost, a nekada čak i bolju uspješnost od Fibonaccijeve hrpe. Treba uzeti u obzir da Fibonaccijeva hrpa daje amortizirana vremena na *deleteMin()* i *decreaseKey()* operacije, te da operacija *decreaseKey()* može imati veliki konstantni faktor, zbog mogućih kaskadnih rezova. Prednost Fibonaccijeve hrpe, u odnosu na binarnu, manifestirala bi se na većim grafovima kada bi asimptotska složenost operacija Fibonaccijeve hrpe imala puno veći utjecaj.

Iako se već na relativno malim grafovima vidi neefikasnost nekih implementacija, na grafovima veličine 10 sve su varijante algoritma imale podjednaka vremena izvršavanja. Zanimljivost koja se pojavila prilikom nekih pokretanja algoritma je brže izvođenje Dijkstrinog algoritma s Fibonaccijevom hrpom na većim grafovima u odnosu na upola manje grafove. Ovakva anomalija je viđena samo na grafovima veličine do 100 vrhova i vjerojatno je posljedica slučajno generiranih grafova koji su zahtijevali manje poziva *decreaseKey()* operacije.

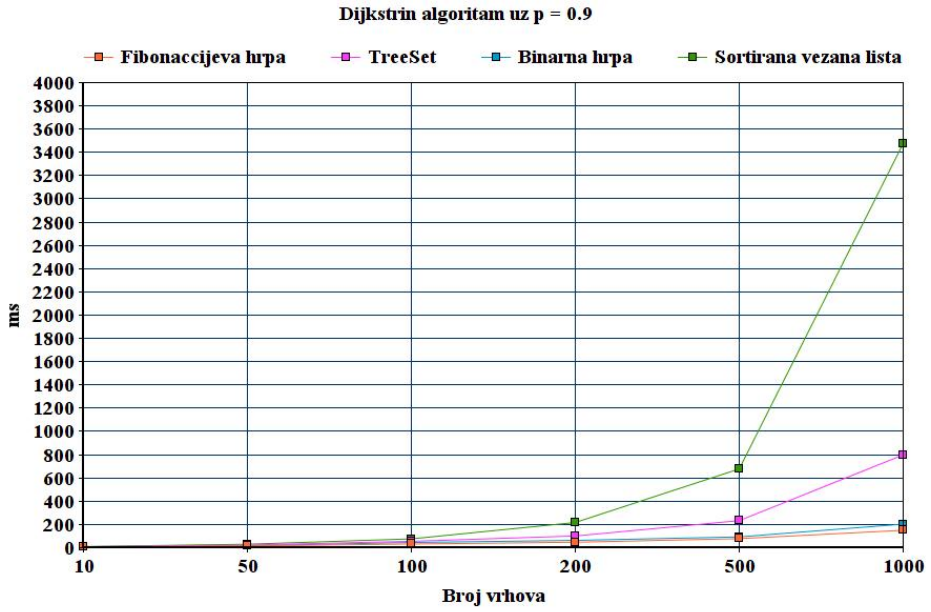
Na slici 4.2 su prikazani podaci dobiveni testiranjem raznih verzija Primovog algoritma, pri čemu, zbog preglednosti, nisu prikazani rezultati varijante algoritma s vezanom sortiranom listom. Podaci su donekle i očekivani očekivani uzme li se u obzir da Primov algoritam ima jednako vrijeme izvršavanja kao i Dijkstrin algoritam. Vremena izvršavanja Primovog algoritma su bila nešto veća u odnosu na Dijkstrin, no Fibonaccijeva hrpa se pokazala malo efikasnijom od binarne hrpe.

4.2 Zaključak

Prioritetni red se pokazao kao bitna struktura za ispravan rad algoritama u više grana računarstva pa se stoga zahtijeva efikasna implementacija prioritetnog reda. U izobilju

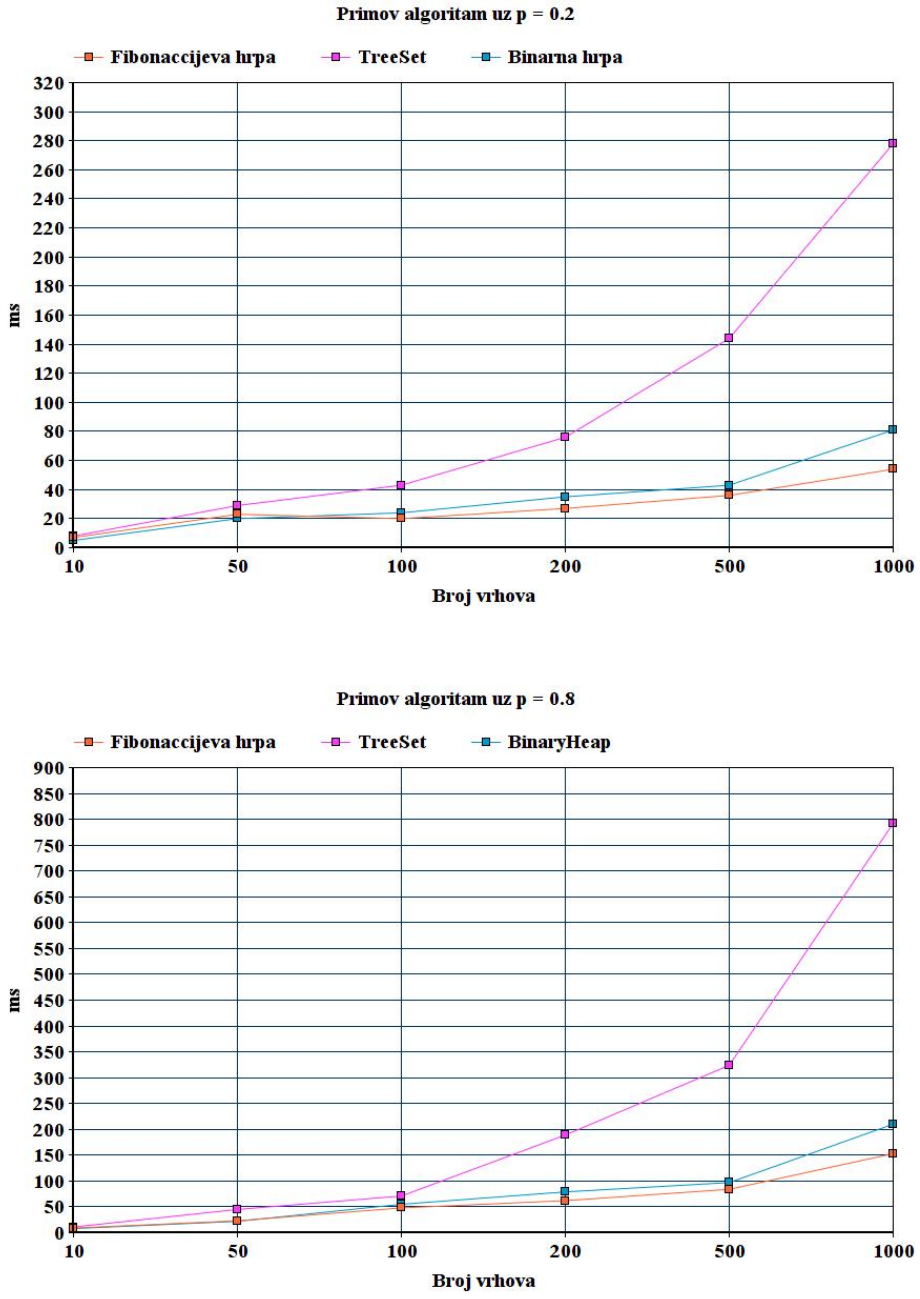


(a)



(b)

Slika 4.1: Prikaz vremena izvršavanja različitih verzija Dijkstrinog algoritma u ovisnosti o veličini grafa.



Slika 4.2: Prikaz vremena izvršavanja različitih verzija Primovog algoritma u ovisnosti o veličini grafa.

kandidata za implementaciju prioritetnog reda, dvije strukture podataka su se posebno izdvojile kao efikasno rješenje. Binarna hrpa garantira $O(\log n)$ vrijeme svih operacija u najgorem slučaju, te uz to nudi jednostavnu implementaciju koja je moguća bez pokazivača, tj. uz korištenje polja. S druge strane je Fibonaccijeva hrpa, koja nudi konstantna amortizirana vremena za operacije ubacivanja elementa i smanjivanja ključa proizvoljnog elementa. Cijena ovakvih vremena operacija je plaćena složenijom implementacijom, što je glavni razlog zbog čega Fibonaccijeva hrpa nije zaživjela u praksi.

Pokazalo se da Fibonaccijeva hrpa ima ogroman učinak na rješavanje problema kombinatorne optimizacije, pri čemu su problem najkraćeg puta i problem minimalnog razapinjućeg stabla dobili efikasnija rješenja u vidu bržih varijanti Dijsktrinog i Primovog algoritma. Iako Fibonaccijeva hrpa ima veliki utjecaj na rad Dijkstrinog algoritma, današnje verzije Dijkstrinog algoritma ne rade na velikim grafovima, već služe kao potprogrami na pretprocesiranim grafovima. Stoga Fibonaccijeve hrpe nisu nužne u implementaciji Dijkstrinog algoritma.

Poboljšanja Fibonaccijeve hrpe su moguća u smjeru stroge Fibonaccijeve hrpe, strukture koja nudi ista vremena operacija kao i Fibonaccijeva hrpa, samo u najgorem slučaju. Najveći fokus treba biti na razvijanju jednostavnije strukture koja će, uz teorijsku, imati i praktičnu svrhu.

Bibliografija

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin i Robert E. Tarjan, *Faster algorithms for the shortest path problem*, Journal of the ACM (JACM) **37** (1990), br. 2, 213–223.

[2] Gerth Stølting Brodal, *Worst-Case Efficient Priority Queues.*, SODA, sv. 96, 1996, str. 52–58.

[3] Gerth Stølting Brodal, George Lagogiannis i Robert E. Tarjan, *Strict fibonacci heaps*, Proceedings of the forty-fourth annual ACM symposium on Theory of computing, ACM, 2012, str. 1177–1184.

[4] Edsger W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische mathematik **1** (1959), br. 1, 269–271.

[5] Robert W. Floyd, *Algorithm 245: Treesort*, Communications of the ACM **7** (1964), br. 12, 701.

[6] Michael L. Fredman i Robert Endre Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM (JACM) **34** (1987), br. 3, 596–615.

[7] V. Jarnık, *About a certain minimal problem*, Práce Moravské Prırodovedecké Spolecnosti **6** (1930), 57–63.

[8] Clifford Stein, T. Cormen, R. Rivest i C. Leiserson, *Introduction to algorithms*, MIT Press Cambridge, MA, 2009.

[9] Robert Endre Tarjan, *Amortized computational complexity*, SIAM Journal on Algebraic Discrete Methods **6** (1985), br. 2, 306–318.

[10] Jean Vuillemin, *A data structure for manipulating priority queues*, Communications of the ACM **21** (1978), br. 4, 309–315.

[11] John William Joseph Williams, *Algorithm-232-heapsort*, Communications of the ACM **7** (1964), br. 6, 347–348.

Sažetak

Fibonaccijeva hrpa je varijanta hrpe, razvijena od strane Fredmana i Tarjana s ciljem poboljšanja Dijkstrinog algoritma najkraćeg puta. Fibonaccijeva hrpa je prva hrpa koja obavlja operacije ubacivanja i smanjivanja ključa u konstantnom amortiziranom vremenu, uz logaritamsko vrijeme brisanja elemenata. S obzirom na utjecaj Fibonaccijeve hrpe na računarstvo, ovaj rad se bavi opisom Fibonaccijeve hrpe i njenog utjecaja na kombinatornu optimizaciju. Provedena je analiza rada Dijkstrinog i Primovog algoritma s Fibonaccijevom hrpom u odnosu na druge moguće implementacije prioritetnog reda.

Summary

Fibonacci heap is a variant of heap, developed by Fredman and Tarjan with the purpose of improving Dijkstra's shortest path algorithm. Fibonacci heap is a first heap that accomplishes insert and decrease key operations in constant amortized time with logarithmic time for deleting the elements. Because of the importance of Fibonacci heap in the field of computer science, this work describes Fibonacci heap and it's influence on the field of combinatorial optimization. Analysis on the influence of Fibonacci heaps in Dijkstra's and Prim's algorithms has been made, in respect to some other possible implementations of priority queue.

Životopis

Marino Lončar rođen je 7. ožujka 1992. u Splitu, gdje je odličnim uspjehom završio osnovnu školu Mertojak i III. gimnaziju. Preddiplomski sveučilišni studij matematike na Matematičkom odsjeku Prirodoslovno–matematičkog fakulteta Sveučilišta u Zagrebu upisao je 2010. godine. Na istom fakultetu 2013. godine dobio je titulu sveučilišnog prvostupnika matematike, te je upisao diplomski studij Računarstva i matematike.