Elektrotehnički fakultet
Univerziteta u Beogradu

Master rad

Analiza i implementacija hip struktura podataka

Kandidat:

Stefan Vitorović

dipl. inž.

Mentor:

dr Milo Tomašević

red. prof

Beograd, septembar 2016.

SADRŽAJ

1. Uvod 4
2. Teorijska analiza hipova 6
  2.1. Binarni hip 8
    2.1.1. Operacije 9
  2.2. Binomijalni hip 12
    2.2.1. Operacije 14
  2.3. Fibonačijev hip 17
    2.3.1. Operacije 18
  2.4. Korišćenje hipova 24
3. Implementacija hipova 27
  3.1. Umetanje elementa 27
  3.2. Nalaženje elementa sa najmanjim ključem 29
  3.3. Brisanje elementa sa najmanjim ključem 30
  3.4. Smanjivanje vrednosti ključa 33
  3.5. Brisanje elementa 35
  3.6. Unija dva hipa 36
  3.7. Implementaciona iskustva 38
4. Metodologija analize 39
  4.1. Razvojna platforma 39
  4.2. Test okruženje 40
    4.2.1. Merenje vremena 40
5. Rezultati analize 41
  5.1. Osnovne operacije 41
    5.1.1. Umetanje elementa 41
    5.1.2. Nalaženje elementa sa najmanjim ključem 43
    5.1.3. Brisanje elementa sa najmanjim ključem 44
    5.1.4. Smanjivanje vrednost ključa 46
    5.1.5. Brisanje elementa 49
    5.1.6. Unija hipova 51
  5.2. Simulacije algoritama 51
    5.2.1. Dijkstra algoritam 51
    5.2.2. Hafmanovi kodovi 52
6. Zaključak 53
7. Literatura 55

1. UVOD

Veliki broj aplikacija zahteva podatke u određenom redosledu, ali ne obavezno u potpuno sortiranom redosledu, odnosno, ne obavezno sve podatke odjednom. Veoma često, aplikacije zahtevaju najveći ili najmanji element, nakon toga se radi unos novih elememata, ili neko brisanje, a nakon toga npr. ponovo element sa najmanjom ili najvećom vrednosti. Idealna struktura podataka za takav rad je prioritetni red. Upotreba prioritetnih redova je slična upotrebi redova (ukljanjanje najstarijeg) ili upotrebi stekova (uklanjanje najmlađeg), ali efikasna implementacija je dosta izazovnija. [1]

Operacije koje se najčešće koriste pri radu sa prioritetnim redovima su unos novih elemenata, brisanje elementa, brisanje elementa sa najmanjom/najvećom vrednosti ključa, umanjenje vrednosti ključa nekog elementa, unija dva prioritetne reda, kao i pronalazak elementa sa najmanjom ili najvećom vrednosti ključa. Gledajući moguće različite načine implementacije ovakve strukture podataka, počevši od upotrebe različitih vrsta nizova, odnosno ulančanih lista, kao struktura podataka koja daje najbolje performanse pokazala se struktura pod nazivom hip (*heap*). [2]

U okviru ovog rada, biće obrađene tri vrste hipa: binarni hip, binomijalni hip i Fibonačijev hip. Svaki od ovih hipova će biti detaljno opisan i obrađen kroz karakteristične operacije. Osnovni cilj rada je merenje odnosno upoređivanje vremenskih perfomansi svake od implementiranih operacija, u zavisnosti od različitog broja ulaznih elemenata. Nakon pojedinačnog upoređivanja operacija, biće urađeno i upoređivanje nekoliko operacija u kombinaciji, kako bi se što bolje simuliralo realno korišćenje prioritetnih redova, odnosno hipova.

U drugom poglavlju je predstavljena teorijska analiza hipova, način na koji se radi klasifikacija hipova, kao i opis izabranih hipova, detaljni opis operacija za svaki od hipova, kao i najčešće primene izabranih hipova. U okviru trećeg poglavlja dat je detaljan prikaz implementacije svake od ovih operacija, za izabrane hipove. Pored opisa implementacija, dat je prikaz implementacija i u okviru delova koda koje su najbitnije za tu operaciju. Poslednji deo tog poglavlja je prikaz implementacionih iskustava.

Četvrto poglavlje se bavi metodologijom analize, odnosno opisom softverske i hardverske platforme na kojima se radila analiza, kao i opisom samog test okruženja. Sledeće poglavlje je vezano za prikaz i analizu rezultata merenja. Prvo će biti prikazani i obrađeni rezultati osnovnih operacija, a nakon toga će biti obrađeno merenje nekoliko uzastopnih operacija.

Šesto poglavlje predstavlja zaključak rada, a u okviru njega su naznačeni i pravci daljeg istraživanja. Sedmo, odnosno poslednje poglavlje sadrži pregled literature i izvora korišćenih prilikom izrade ovog rada. Literatura je navedena u IEEE formatu preporučenom u [22][22], a tamo gde je bio dostupan postavljen je i odgovarajući link na internetu.

2. TEORIJSKA ANALIZA HIPOVA

Hip je struktura podataka bazirana na stablu, koja zadovolja sledeće hip osobine: ako je ključ A nekog elementa, roditelj elementa sa odgovarajućim ključem B, tada je redosled koji se javlja između bilo koja dva elementa koja su u odnosu roditelj – dete, isti kao redosled između elementa sa ključem A i elementa sa ključem B. Na osnovu toga, osnovna podela hipova može biti na *max heap* i *min heap*. U okviru *max* hipa, ključ elementa koji je roditelj jednom ili više elemenata je uvek veći ili jednak od ključeva elemenata dece, a najveći ključ je onaj koji je *root* (koren stabla). Suprotno od toga je *min heap,* gde je koren stabla element sa najmanjim ključem, a roditelj je uvek manji ili jednak od svih ključeva elemenata dece. U okviru ovog rada, obrađivaće se isključivo *min* hip. [2]

Operacije koje su najznačajnije u radu sa *min* hipom, odnosno one koje će biti obrađene u okviru ovog rada su:

- ***insert (k, e)*** – umetanje elementa *e* sa ključem *k*
- ***min()*** – čitanje elementa sa najmanjom vrednošću ključa
- ***extract_min()*** – čitanje, a nakon toga i uklanjanje elementa sa najmanjom vrednošću ključa
- ***decrease_key(e, k)*** – zamena vrednosti ključa elementa *e* sa ključem *k.* Osnovni uslov je da ključ *k* ima manju vrednost od ključa trenutnog ključa elementa *e*
- ***delete (e)*** – brisanje elementa *e*
- ***union (heap1, heap2)*** – unija dva hipa

U zavisnosti od načina implementacije, odnosno da li se elementi hipa nalaze u okviru niza ili im se međusobno pristupa pomoću pokazivača, kao i od samog načina organizacije elemenata, postoji veći broj različitih vrsta hipova. Hipovi koji se najčešće koriste, odnosno svoju primenu mogu naći u nekom od algoritama gde su potrebne određene karakteristike prioritetnih redova su:

- **Binarni hip** – osnovna verzija hipa, i za implementaciju najjednostavnija. Garantovane vremenske performanse operacija *insert, extract_min i decrease_key* su *O(log n).*
- **D – array hip –** generalizacija binarnog hipa, u okviru koje umesto čvorova sa dva deteta se javljaju čvorovi sa d dece. Prilikom *decrease_key* operacije, ovaj način raspodele daje dosta bolje performanse od binarnog hipa, međutim performanse prilikom *delete* operacije su lošije.
- **Binomijalni hip –** predstavlja skup binomijalnih stabala, tako da svako binomijalno stablo zadovoljava min hip uslov. Pored tog svojstva, drugo važno svojstvo po kojem je ova vrsta hipa prepoznatljiva je da svako stablo u okviru korena binomijalnog hipa ima različitu visinu. Prosečna vremenska složenost *insert* operacija je linearna, dok je za ostale operacije *O(log n).*
- **Fibonačijev hip** – najčešća upotreba ovog hipa je u okviru Dijkstra algoritma, odnosno u okviru algoritama gde je izražena operacija *decrease_key* s obzirom da ova vrsta hipa tu operaciju izvršava u konstantnoj vremenskoj složenosti. Za operacije *extract_min* i *delete*, vremeneska složenost je *O(log n).*
- **Pairing hip –** smatra se kao jednostavnija varijanta Fibonačijevog hipa. Implementacija ovog hipa je relativno jednostavna, a prosečne performanse su dosta dobre. Vremenska složenost *extract_min* operacija je *O (log n),* dok operacije *min*, *union* i *insert* imaju kontantnu vremensku složenost. Ova vrsta hipa se često upotrebljava u okviru Primovog algoritma za minimalna obuhvatna stabla. **Error! Reference source not found.**
- **2 – 3 hip** – ova vrsta hipa je osmišljena od strana Tadao Takaoka 1999 godine i slična je Fibonačijevom hipu. Pored Fibonačijevog hipa, koristi i ideju koja je u osnovi 2-3 stabala, i po tome je ovaj hip i dobio ime. **Error! Reference source not found.**

U tabeli 2.1 dat je prikaz vremenskih složenosti osnovnih operacija kod nabrojanih hipova. Kao što se može videti, uzimanje elementa sa najmanjom vrednosti i brisanje elementa ima logaritamsku vremensku složenost kod svih vrsta hipova, dok ostale operacije, najbolje složenosti imaju kod Fibonačijevog hipa (konstantna vremenska složenost kod svih ostalih operacija).

	Binarni	Binomijalni	Fibonačijev	Pairing	2-3
insert	O(log n)	O(log n)	O(1)	O(1)	O(1)
min	O(1)	O(log n)	O(1)	O(1)	O(1)
extract_min	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
decrease_key	O(log n)	O(log n)	O(1)	O(log n)	O(1)
delete	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
union	O(m log k)	O(log n)	O(1)	O(1)	/

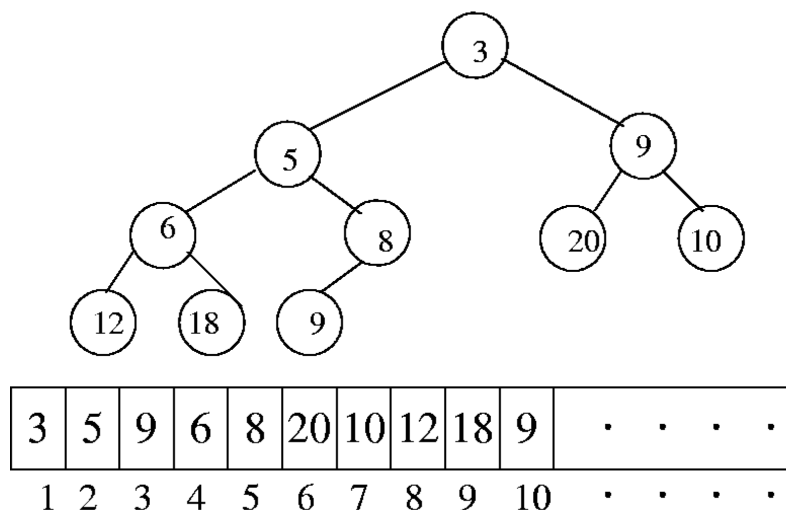
Tabela 2.1 – Prikaz vremenskih složenosti osnovnih operacija kod najpoznatijih hipova

U okviru ovog poglavlja detaljno će biti opisane tri vrste hipa koje su izabrane, opisani algoritmi za svaku od operacija, kao i objašnjena najčešća primena i razlozi korišćenja. Binarni hip je najčešće korišćena i za implementaciju najjednostavnija vrsta hipa, koja zbog načina generisanja, odnosno smeštanja elemenata veliku primenu ima u nekoliko algoritama, od kojih je svakako najpoznatiji *heapsort* [4]. Jedna od velikih mana binarnog hipa je linearna vremenska složenost pri operaciji spajanja dva hipa, što je efikasno rešeno u okviru binomijalnog, a posebno Fibonačijevog hipa, kod kojih ta operacija ima logaritamsku, odnosno konstantnu vremensku složenost.

Binomijalni hip, pored svoje prednosti u odnosu na binarni u operaciji spajanja dva hipa, značajan je i što po strukturi, predstavlja osnovu za Fibonačijev hip. Kao hip koji se po vremenskim složenostima, posebno kod operacija umetanja novog elementa, umanjenja vrednosti ključa i uniji dva hipa ističe, Fibonačijev hip je predstavljao logičan izbor pri odabiru hipova za analizu.

2.1. Binarni hip

Binarni hip je kompletno binarno stablo sa elementima u parcijalno uređenom redosledu, tako da je svaki ključ nekog elementa koje je roditelj manji od ključa elementa levog , odnosno desnog deteta.



Slika 2.2 – Izgled binarnog hipa [1]

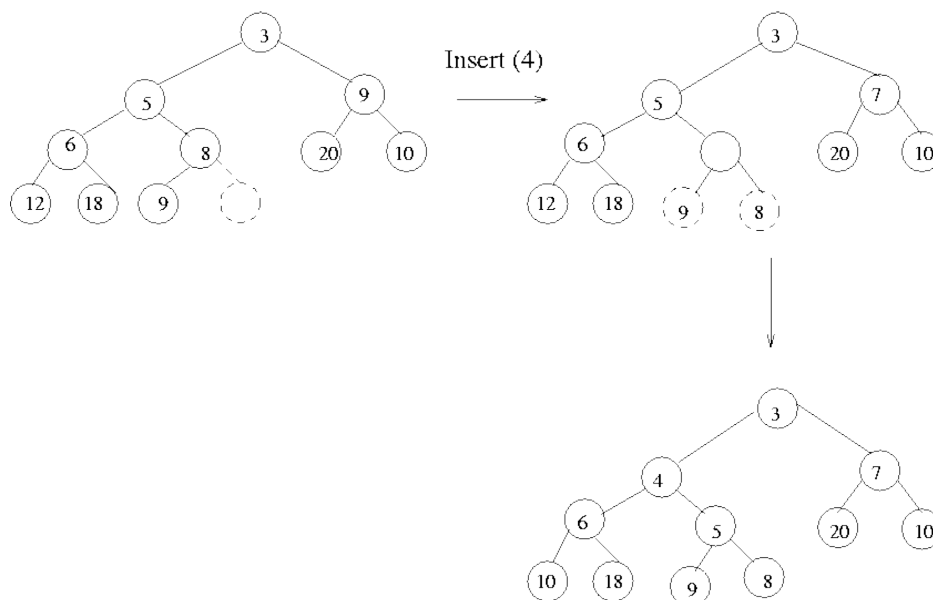
S obzirom da je u pitanju kompletno binarno stablo, elementi mogu biti smešteni u niz. Ako je pozicija nekog elementa *i*, pozicija levog deteta je *2i,* a desnog deteta $2i + 1$. Na isti način, može se odrediti da se roditelj od nekog elementa nalazi na $\left\lfloor \frac{i}{2} \right\rfloor$. Zahvaljujući ovakvoj strukturi, hip sa visinom k može imati između 2^k i $2^{k+1} - 1$ elemenata. Takođe, hip sa *n* elemenata ima visinu $\lfloor log_2 n \rfloor$. [2]

2.1.1. Operacije

Umetanje elementa

Za umetanje elementa sa ključem *x*, u hip sa koji trenutno sadrži *n* elemenata, u prvom koraku, potrebno je napraviti staviti taj element na poziciju $n + 1$ i nakon toga proveriti da li on zadovoljava *min* svojstvo hipa da je ključ od elementa roditelja manji ili jednak ključu od elementa deteta. Ako zadovoljava, pronađeno je odgovarajuće mesto za taj element, i umetanje u hip je završeno. U suprotnom, tek umetnuti element je potrebno zameniti sa elementom roditelja (zamena na gore), i nakon toga takođe proveriti da li novo stanje odgovara svojstvu hipa. Na osnovu toga, može se zaključiti da se zamena vrši dok se na nađe stanje hipa da odgovara osobinama hipa, odnosno, u krajnjem slučaju, to znači da će se tek umetnuti element naći u korenu stabla hipa.

S obzirom na osobine operacije umetanja novog elementa da se čvor može menjati celom visinom stabla, vremenska složenost ovog algoritma u najgorem scenariju je *O(h),* gde *h* predstavlja visinu hipa, odnosno, vremenska složenost je *O(log n).*



Slika 2.3– Primer operacije umetanja novog elementa u binarni hip [1]

Nalaženje elementa sa najmanjim ključem

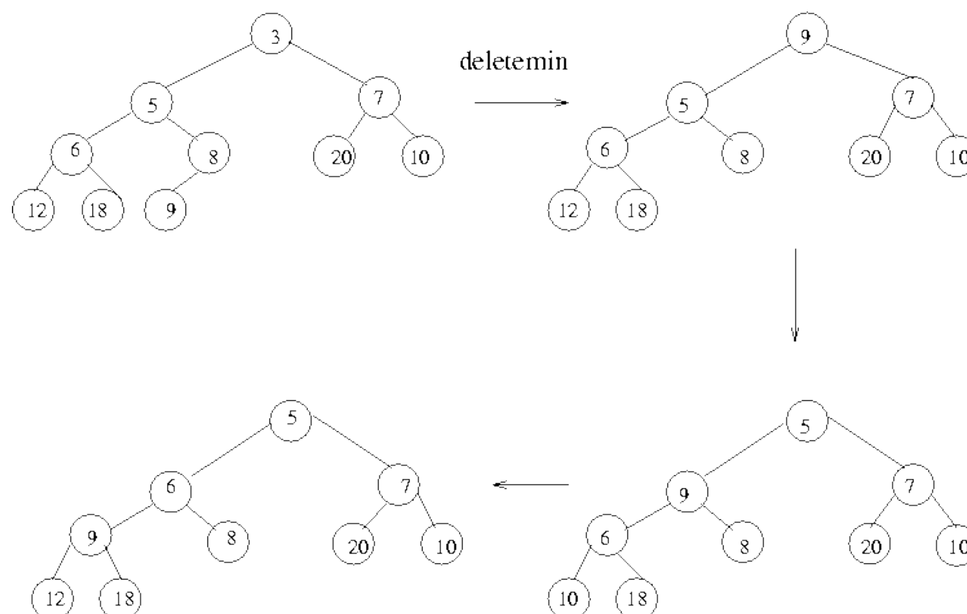
Na osnovu opisanog postupka umetanja elementa, kao i same osobine hipa, može se zaključiti da se element sa minimalnim ključem uvek nalazi u korenu hipa.

Vremenska složenost za ovu operaciju je konstantna, odnosno *O(1).*

Brisanje elementa sa najmanjim ključem

Prvi korak u okviru ove operacije je fizičko uklanjanje korena, odnosno elementa sa najmanjim ključem. Nakon toga, tu poziciju zauzima poslednji element iz hipa. Prebacivanjem poslednjeg element u koren, potrebno je proveriti da li u tom trenutku hip zadovolja svojstva. Ako zadržava, brisanje elementa sa najmanjim ključem je završeno, dok u suprotnom, radi se zamena novog korena sa elementom deteta koji ima manji ključ (zamena na dole) [1]. Kao što je i očekivano, nakon zamene dolazi do provere da li trenutno stanje zadovoljava svojstva hipa. Zamena na dole i provera se rade sve dok se ne nađe odgovarajuće mesto za taj čvor.

Sama operacija brisanja najmanjeg ključa je konsntanta, međutim, kao i kod operacije umetanja novog elementa, čvor može da menja poziciju čitavom visinom stabla, tako da je složenost ove operacije logaritamska *O(log n).*



Slika 2.4 – Brisanje elementa sa najmanjim ključem [1]

Smanjivanje vrednosti ključa

U okviru ove operacije, ne razmatra se postupak pronalaženja elementa sa ključem čija se vrednost želi smanjiti, već se smatra da je element već nađen i radi se umanjenje ključa. Prvi korak nakon umanjenja ključa je provera da li hip u tom stanju zadovoljava svojstva. Ako zadovoljava, postupak se završava, u suprotnom, ponavlja se proces kao kod operacije umetanja elementa , gde se radi "zamena na gore", odnosno ključ se propagira ka korenu stabla. Prilikom svake od ovih zamena, radi se i provera da li hip u tom trenutku zadovoljava *min* hip svojstva.

S obzirom da ključ koji se menja može biti i na poslednjem nivou stabla, a propagiranja se može obavljati do korena, vremenska složenost ovog algoritma je logaritamska, odnosno *O(log n).*

Brisanje elementa

Kao i u okviru operacije smanjivanja vrednosti ključa, ni ovde se ne razmatra postupak pronalaženja elementa sa ključem koji se briše, već se smatra da je element već nađen i radi se postupak brisanja. Nakon fizičkog uklanjanja elementa iz stabla, na mesto gde je bio obrisani stavlja se poslednji element iz hipa i proverava se da li hip u tom stanju zadovoljava svojstva. Ako zadovoljava, postupak se završava, u suprotnom, ponavlja se proces kao kod operacije brisanja elementa sa najmanjim ključem, gde se radi "zamena na dole", odnosno ključ se propagira ka listovima stabla. Prilikom svake od ovih zamena, radi se i provera da li hip u tom trenutku zadovoljava *min* hip svojstva.

S obzirom da ključ koji se menja može biti i u korenu, a propagiranja se može obavljati do poslednjeg nivoa stabla, vremenska složenost ovog algoritma je logaritamska, odnosno *O(log n).*

Unija hipova

U okviru ove operacije, prvi korak je kopiranje svih elemenata hipa koji se ubacuje, u hip u koji se ubacuje. Ako hip u koji se ubacuje ima veličinu *k*, elementi drugog hipa će se smestiti na lokacije od *k+1*. Nakon toga, radi se ponovno kreiranje hipa, na način da se kreće od poslednjeg elementa hipa, radi se provera ključa sa ključem roditelja i u zavisnosti od rezultate te provere radi se zamena. Postupak se ponavlja za sve elemente hipa.

Vremenska složenost ovakvog algoritma zavisi od provere ključa koji se gleda sa ključem roditelja, a s obzirom da se ta provera mora obaviti kroz sve elemente rezultujućeg hipa, složenost ove operacije je *O (m log k),* gde *m* predstavlja veličinu drugog hipa koji se ubacuje u hip veličine *k* u koji se ubacuje.

2.2. Binomijalni hip

Binomijalni hip je skup binomijalnih stabala. Da bi se ispravno definasao i razumeo binomijalni hip, potrebno je prvi definisati binomalno stablo.

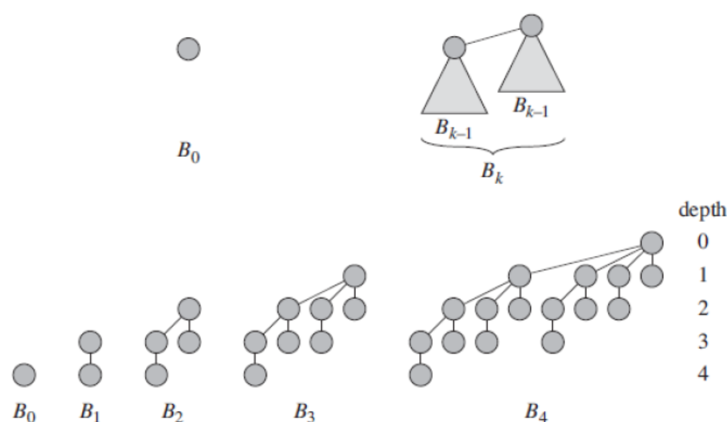
Binomijalno stablo

Binomijalno stablo je uređeno stablo definisano rekurzivno tako da [3]

- B_o sadrži jedan čvor
- za svako $k > 0$, $\mathrm{B_k}$ sadrži dva B_{k-1} stabla međusobno povezano tako da je koren jednog stabla levo dete korena drugog.

Osnovna svojstva binomijalnog stabla su:

- visina stabla je *k*
- ima tačno 2^k čvorova
- ima tačno $\binom{k}{i}$ čvorova na dubinama $i = 0, 1, 2,..., k$
- koren ima stepen *k*
- ako su deca korena numerisani sa leva na desno sa $k - 1$, $k - 2,...$, 0, tada je dete i koren podstabla B_i



Slika 2.5 – Prikaz binomalnog stabla [3]

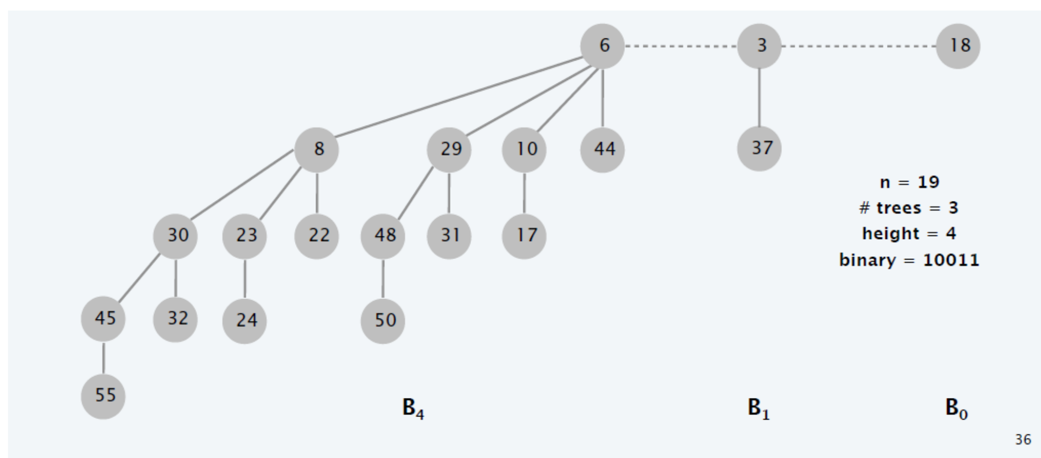
Binomijalni hip

Binomijalni hip H je skup binomijalnih stabala koji zadovoljava sledeća svojstva:

- svako binomijalno stablo u okviru hipa H zadovoljava *min heap* svojstvo – ključ nekog čvora je manji ili jednak od čvora deteta
- postoji 0 ili 1 binomijalno stablo reda k

Na osnovu ovih generalnih svojstava, binomijalni hip H sa n čvorova ima sledeće karakteritike:

- čvor koji sadrži minimalni ključ je koren $B_o, B_1, \dots, ili\ B_k$ binomijalnog stabla
- sadrži binomijalno stablo B_i, ako je b_i = 1, gde $b_k \dots b_2 b_1 b_0$ binarna reprezentacija broja n
- postoji <= $\lfloor log_2 n \rfloor$ + 1 binomijalno stablo
- visina je <= $\lfloor log_2 n \rfloor$



Slika 2.6 – Prikaz binomijalnog hipa sa n čvorova [18]

Jedna od karakteristika binomijalnog hipa, koja se može se videti i sa slike 3.5 je da su koreni binoijalnih stabala povezani, i to u jednostruko ulančanu listu, koja se naziva lista korena hipa *(root*

list). Ovoj listi, odnosno samom hipu *H* se pristupa sa pokazivačem *head[H]* koja pokazuje na prvi koren ove liste. Ako binomijalni hip nema elemenata, vrednost *head[H]* je *null*.

Svaki čvor binomijalnog hipa sadrži ključ kao i dodatne informacije koje zavise od pozicije čvora u okviru hipa. Kao dodatne informacije, mogu se javiti pokazivač na čvor roditelja, pokazivač na najlevlje dete, pokazivač na brata čvora (koji se nalazi sa njegove desne strane). Ako se čvor *x* nalazi u listi korena binomijalnog hipa, pokazivač na roditelja je *null*, ako čvor *x* nema decu, pokazivač na čvor deteta je *null*, a ako čvor nema braću koja se nalaze sa njegove desne strane, pokazivač na čvor brate je *null*. Pored ovih informacija, svaki čvor ima i informaciju stepen(*degree*) koja govori koliko svaki čvor ima dece.

2.2.1. Operacije

Nalaženje čvora sa najmanjim ključem

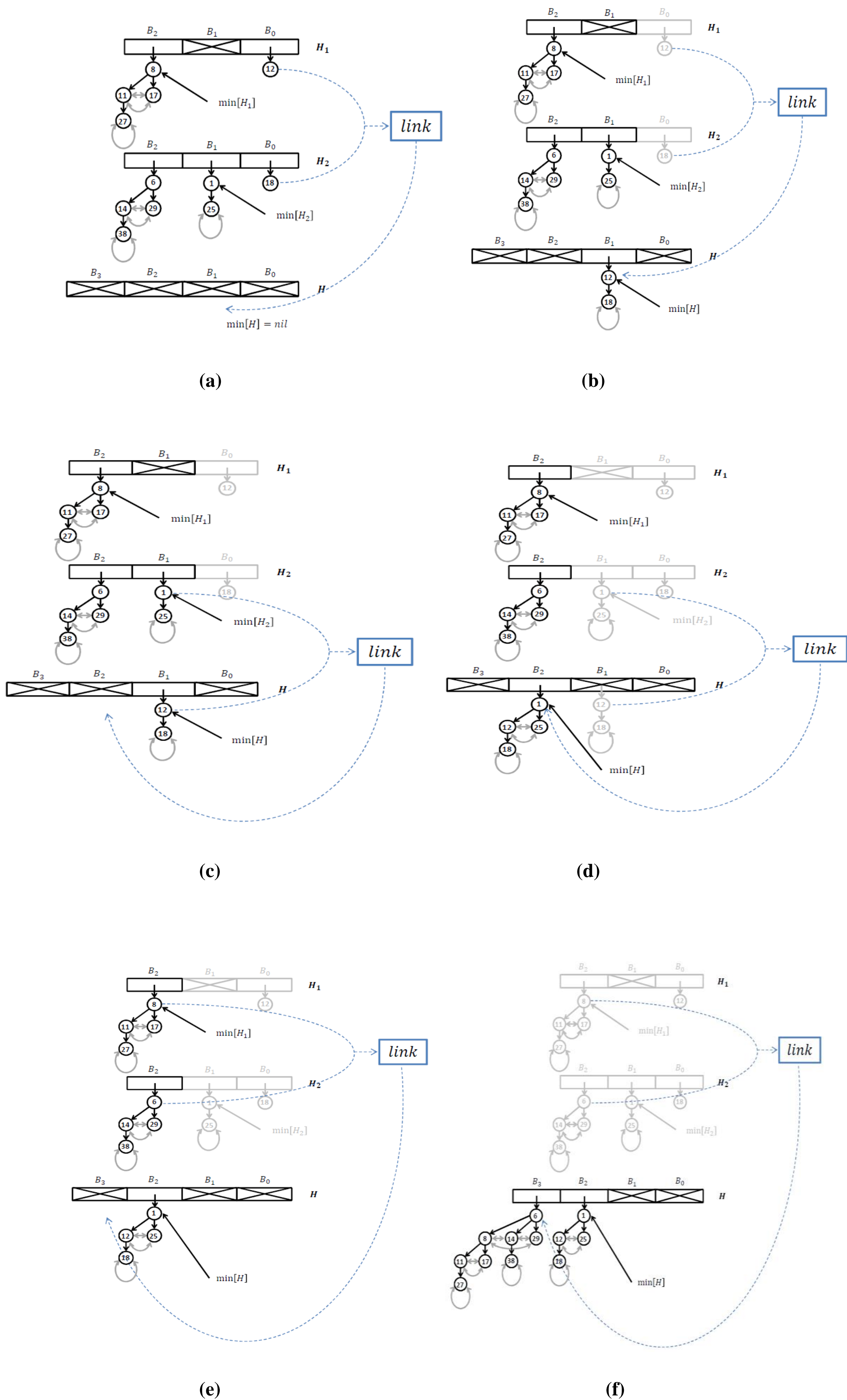
S obzirom da se po svojstvima binomijalnog hipa, svaki hip sastoji od binomijalnih stabala gde svako stablo ima *min heap* svojstva, odnosno u korenu svakog stabla se nalazi minimalni ključ za to stablo, može se zaključiti da se čvor sa minimalnim ključem nalazi u listi korena stabala *(root list).* Prema tome, najmanji ključ binomijalnog hipa se dobija pronalaženjem najmanjeg ključa u jednostruko ulančanoj listi.

Po karakteristikama binomijalnog hipa, postoji maksimalno $\lfloor log_2 n \rfloor$ + 1 stabala, odnosno isto toliko korena od tih stabala, tako da je vremenska složenost za binomijalni hip sa *n* čvorova logaritamska *O(log n).*

Unija hipova

Operacija unije (spajanja) dva binomijalna hipa se koristi kao podrutina, odnosno deo svih preostalih operacija u radu sa binomijalnim hipom, i zbog toga je ova operacija predstavljenja pre ostalih.

U okviru opisa binomijalnog hipa, navedeno je da se svaki binomijalni hip može predstaviti binarnom reprezentacijom broja *n* koji predstavlja ukupan broj čvorova u okviru tog hipa. Unija dva hipa se može posmatrati i kao sabiranje binarnih reprezentacija hipa H_i i hipa H_j. Inicijalno, rezultujući hip *H* je prazan, a početak sabiranja počinje od bita najmanje težine i ide ka bitu najveće težine.



Slika 2.7 – Prikaz unije dva binomijalna hipa [19]

Vremenska složenost algoritma unije dva binomijalna hipa u najgorem slučaju je *O(log n)* gde *n* predstavlja ukupan broj čvorova u rezultujućem hipu *H*. Delimično poboljšanje performansi je moguće ako bi se hip sa manjim ili jednakim brojem elemenata dodavao direktno na drugi hip. I tada bi vremenska složenost bila *O(log n)* ali u prosečnom slučaju sa malo boljim vremenom.

Umetanje elementa

Ova operacija se sastoji iz dva koraka. U prvom, radi se kreiranje novog binomijalnog hipa sa samo jednim čvorom (čvor koji se ubacuje), a u drugom koraku, radi se operacija unije, gde je prvi binomijalni hip upravo kreirani sa čvorom koji se ubacuje, a drugi hip onaj u koji se želi umetanje novog elementa.

Vremenska složenost ovog algoritma se isto može podeliti u dva dela, tako da kreiranje hipa ima složenost *O(1)*, a unija hipova složenost *O(log n)*. Prema tome, sama složenost ovog algoritma je *O(log n)*.

Smanjivanje vrednosti ključa

Operacija smanjivanja vrednosti ključa nekog čvora se kod binomijalnog hipa odvija na isti način kao ta operacija kod binarnog hipa. Ukratko, čvor sa novim ključem se propagira ka vrhu binomijalnog stabla u kojem se nalazi, sve dok ne dođe do roditelja koji je manji od vrednost tog novog ključa, odnosno dok ne dođe do liste korena hipa.

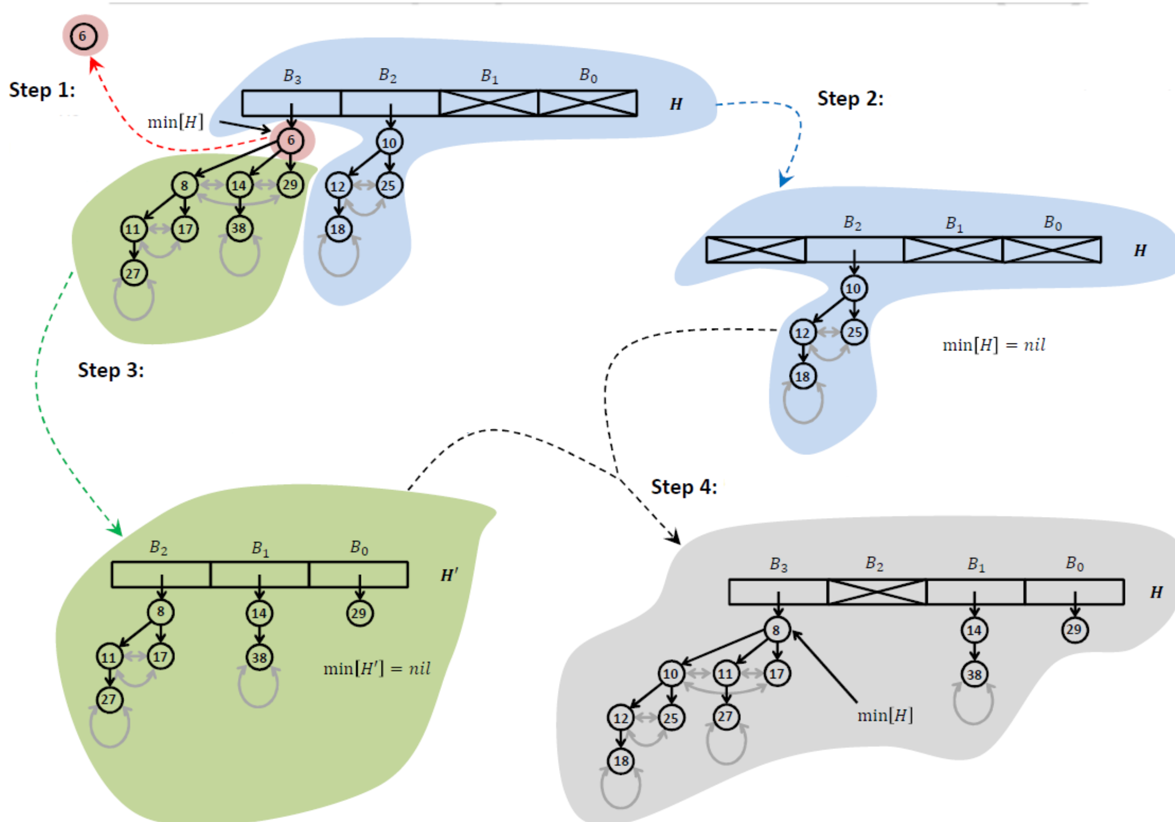
Vremenska složenost je ista kao kod binarnog hipa, odnosno, ona je *O(log n)*.

Brisanje čvora sa najmanjim ključem

Operacija brisanja čvora sa najmanjim ključem kod binomijalnog hipa se može predstaviti u četiri koraka:

1. Pronalazak i brisanje iz liste korena binomijalnih stabala *(root list)* čvora sa najmanjim ključem.
2. Brisanje binomijalnog stabla koje je sadržalo čvor sa najmanjim ključem. Novi hip se može predstaviti kao hip H .
3. Pravljanje hipa H' od dece čvora koji je najmanji u hipu i koji je izbrisan
4. Unija hipa H i H'

Vremenska složenost za korake (1), (3) i (4) je *O(log n),* za korak (2) je *O(1)*, tako da je ukupna vremenska složenost ove operacije *O(log n)*.



Slika 2.8 – Brisanje čvora sa najmanjim ključem [19]

Brisanje čvora

Ova operacija se sastoji iz dva dela: umanjenja vrednosti ključa koji se briše na minimalnu vrednost, tako da taj čvor sigurno postane minimalni čvor u celom hipu, kao i operacije brisanja čvora sa najmanjim ključem.

Obe ove operacije imaju vremensku složenost *O(log n),* tako da je i ukupna vremenska složenost ovog algoritma *O(log n),*

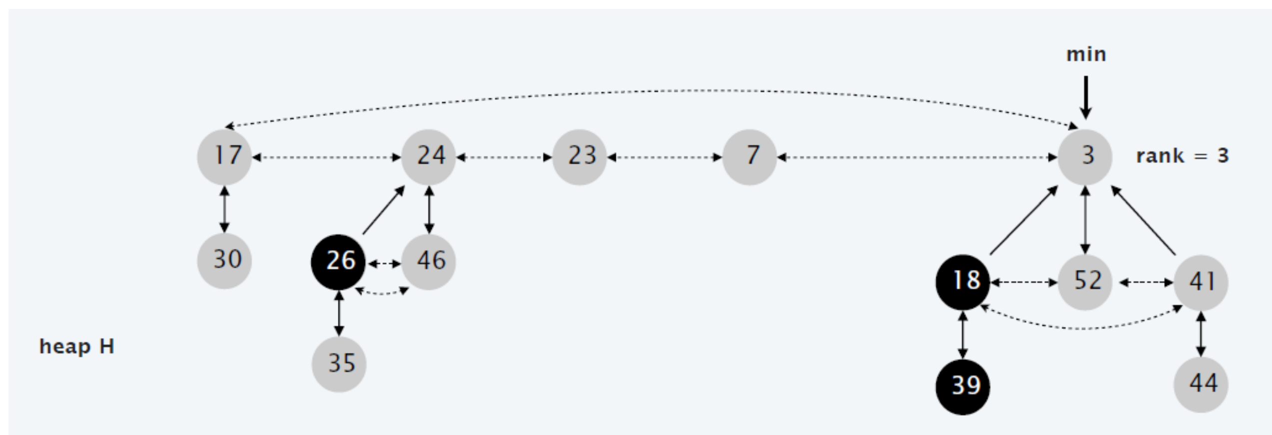
2.3. Fibonačijev hip

Fibonačijev hip je kolekcija korenih stabala koja zadovoljavaju *min-heap* svojstva. Kao što se može videti po definiciji, ova vrsta hipa je dosta slična binomijalnom hipu, međutim, sama svojstva su manje stroža u odnosu na binomijalni hip. Razlike između ove dve vrste hipa se najlakše uočavaju u okviru osnovnih operacija, kao npr. da binomijalni hip radi konsolidaciju stabala nakon svakog umetanja novog elementa, ili prilikom operacije umanjenje vrednosti elementa, radi se zamena sa čvorem roditelja. Za razliku od binomijalnog, Fibonačijev hip konsolidaciju stabala radi tek nakon prvog brisanja najmanjeg elementa, a umanjenje vrednosti elementa se radi odsecanjem čvora koji se umanjuje i stavljanjem u listu korena.

Svaki čvor koji se nalazi u fibonačijevom hipu, može da sadržu sledeće delove (vrednosti i pokazivače):

- ključ čvora
- pokazivač na čvor roditelja, u slučaju da taj čvor ima roditelja, u suprotnom vrednost tog pokazivača je *null*
- pokazivač na jedan od čvorova dece, u slučaju da taj čvor nema čvor deteta, vrednost tog pokazivača je *null*
- pokazivač na čvor levog i desnog brata; ovo znači da su čvorovi koji su braća uvezani u dvostruko ulančanu listu; ako čvor *x* nema braće, onda važi relacija *x.levi_brat* = *x.desni_brat* = *x*; koreni svih stabala su takođe vezani u dvostruktu ulančanu listu
- stepen čvora koji sadrži informaciju koliko dece ima čvor koji se gleda
- *mark* atribut, koji može imati vrednost *true* odnosno *false*

Pored ovih delova koje može da ima svaki čvor, na nivou hipa postoje i pokazivač *min* koji pokazuje na čvor čiji ključ ima najmanju vrednost u celom hipu, kao i atribut *broj_čvorova* koji prikazuje ukupan broj čvorova u hipu. Stavljanje čvorova koji su braća u dvostruko ulančanu listu, donosi dve velike prednosti ovakvoj strukturi podataka: ubacivanje novog elementa na bilo koju lokaciju ima konstantnu vremensku složenost *O(1)* i spajanje dve dvostruko ulančane liste takođe ima *O(1)* vremensku složenost.



Slika 2.9 – Prikaz Fibonačijevog hipa [20]

2.3.1. Operacije

Umetanje elementa

Operacija umetanja elementa kod Fibonačijevog hipa se sastoji iz dva koraka:

1. Umetanja novog elementa u dvostruko ulančanu listu
2. U slučaju da je ključ tog čvora koji je u upravo umetnut manji od ključa do tada minimalnog čvora hipa, radi se prebacivanje *min* pokazivača sa minimalnog čvora na tek ubačeni.

Vremenska složenost za oba koraka je *O(1),* tako da je vremenska složenost za ovu operaciju *O (1).*

Nalaženje čvora sa najmanjim ključem

Kao što je objašnjeno u delu o opisu Fibonačijevog hipa, pokazivač *min* pokazuje na čvor sa najmanjom vrednosti ključa, tako da se ta najmanja vrednost dobija uzimanjem ključa iz tog čvora.

Vremenska složenost je konstantna, odnosno *O(1).*

Unija hipova

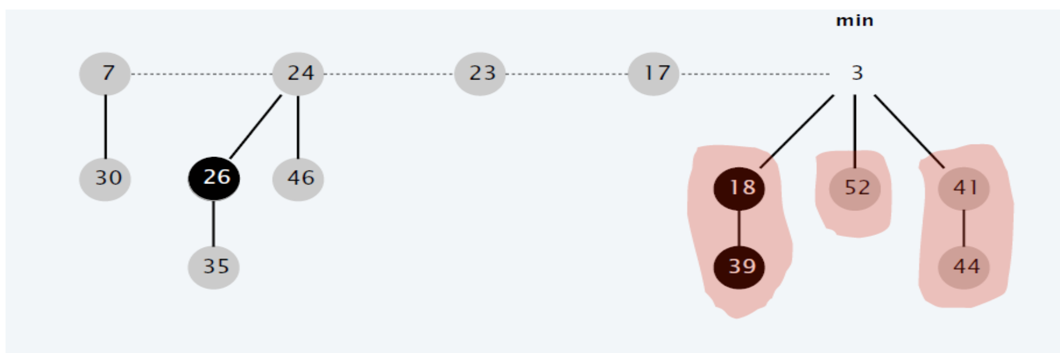
Operacija unije hipova se svodi na povezivanje *root* listi hipova, odnosno preciznije, povezivanje dve dvostruku ulančane liste. Vremenska složenost je *O (1).*

Brisanje čvora sa najmanjim ključem

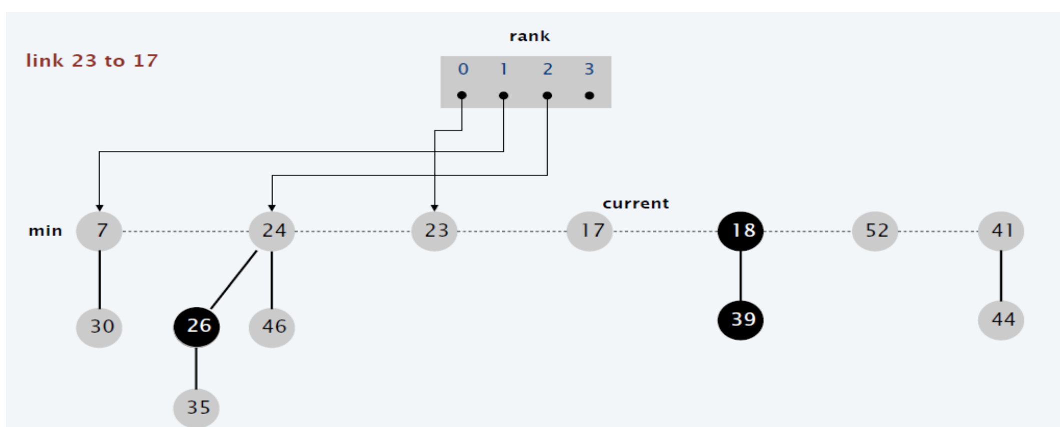
Brisanje čvora sa najmanjim ključem je i najsloženija operacija u radu sa Fibonačijevim hipom. Prvi korak je samo brisanje čvora, stavljanje dece tog najmanjeg čvora u *root* listu hipa i prebacivanje *min* pokazivača na trenutno najmanji čvor u hipu. Drugi korak je mnogo komplikovaniji, jer je cilj da se nakon tog koraka, u okviru hipa ne nađu dva stabla sa istim stepenom. Taj korak se zove i konsolidacija *root* liste.

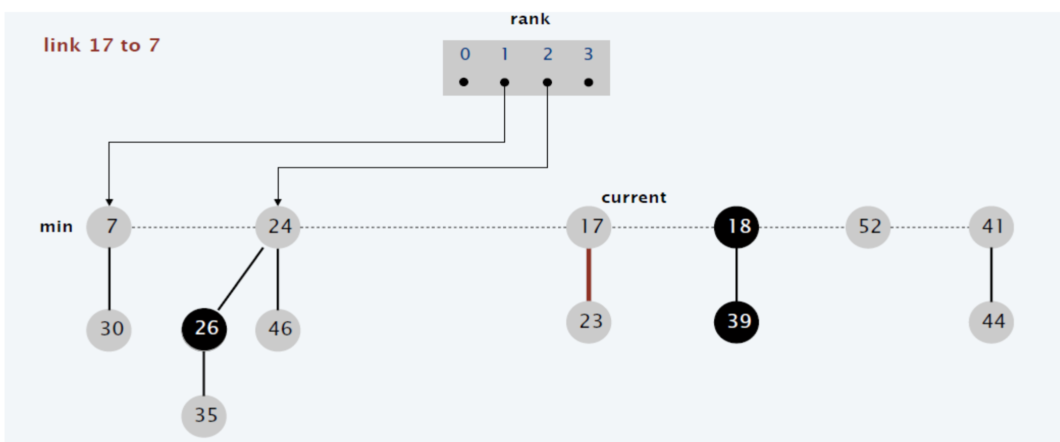
Prilikom konsolidacije, kao pomoćna struktura javlja se niz sa nazivom *rank*, čija je veličina određena gornjom granicom maksimalnog stepena čvora hipa, a matematički je određeno da je ta granica *log n.* Sam proces konsolidacije se obavlja tako što se svaki čvor iz *root* liste pokuša ubaciti u *rank* niz, gde je mesto u nizu na koje se pokušava ubacivanje određeno stepenom tog čvora. Ako je mesto u nizu prazno, odnosno ne pokazuje ni na jedan čvor, dolazi do ubacivanja pokazivača na trenutni čvor u *rank* niz, dok u suprotnom, radi se spajanje po *min* hip svojstvu tekućeg čvora sa tim čvorom iz *rank* niza. Spajanjem dva čvora dolazi do povećanja stepena tog novog čvora, tako da se pokušava taj novi čvor ponovo ubaciti u *rank* niz. Ovaj postupak se ponavlja dok se ne prođu svi čvorovi iz *root* liste. Nakon toga se radi rekonstrukcija hipa na osnovu *rank* niza.

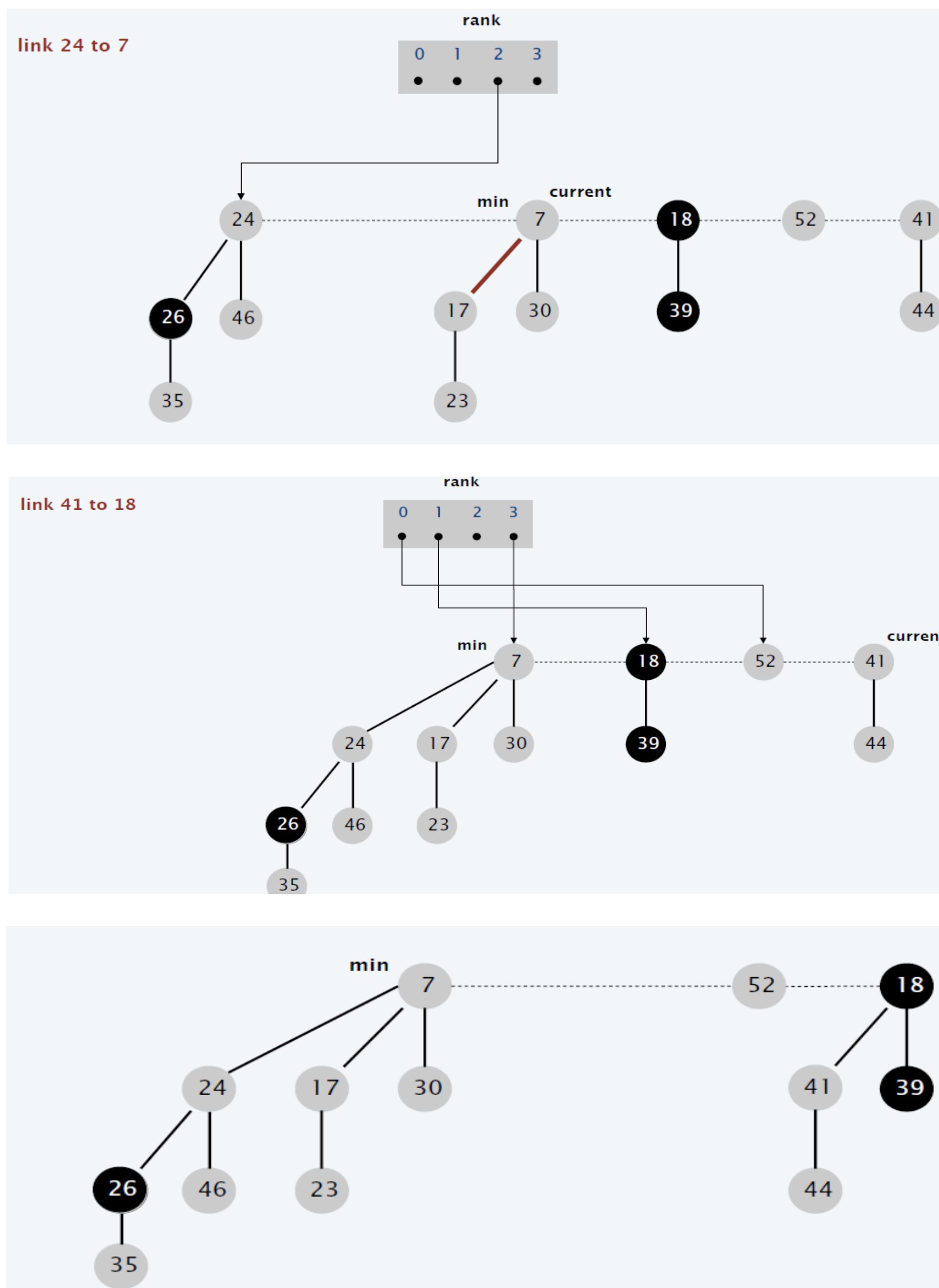
Vremenska složenost ovog algoritma je određena gornjom granicom maksimalnog stepena čvora, a to je *O(log n).*


min
7
24
23
17
3
30
26
46
18
52
41
35
39
44

min
7
24
23
17
18
52
41
30
26
46
39
44
35


link 23 to 17
rank
0 1 2 3
current
min
7
24
23
17
18
52
41
30
26
46
39
44
35


link 17 to 7
rank
0 1 2 3
current
min
7
24
17
18
52
41
30
26
46
23
39
44
35

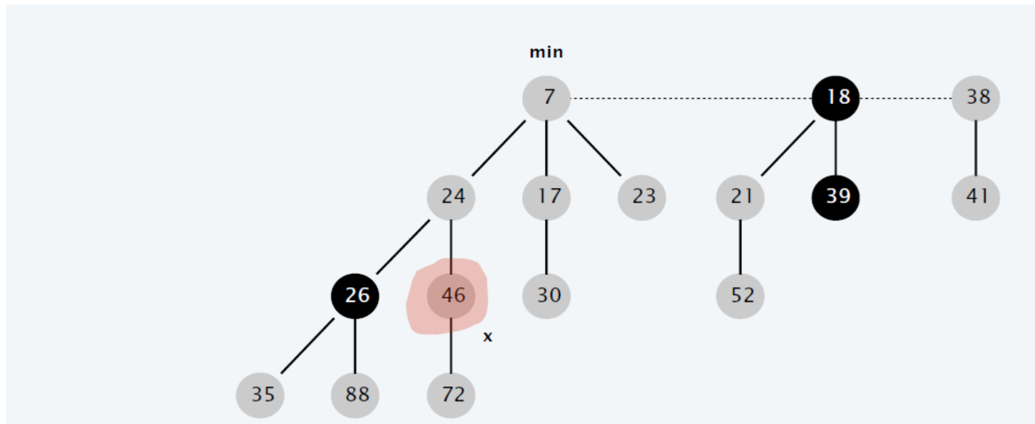


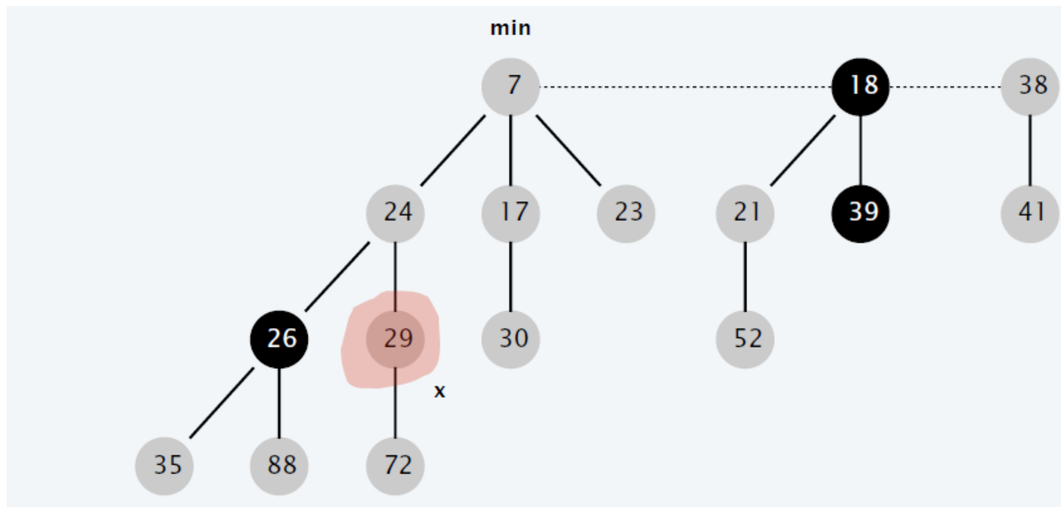
Slika 2.10 – Prikaz brisanja čvora sa najmanjim klučem u Fibonačijevom hipu [20]

Smanjivanje vrednosti ključa

Operacija smanjivanje vrednosti ključa, u zavisnosti od vrednosti na koju se smanjuje i položaja čvora čiji se ključ smanjuje ima nekoliko različitih slučajeva na koji način se operacija obavlja:

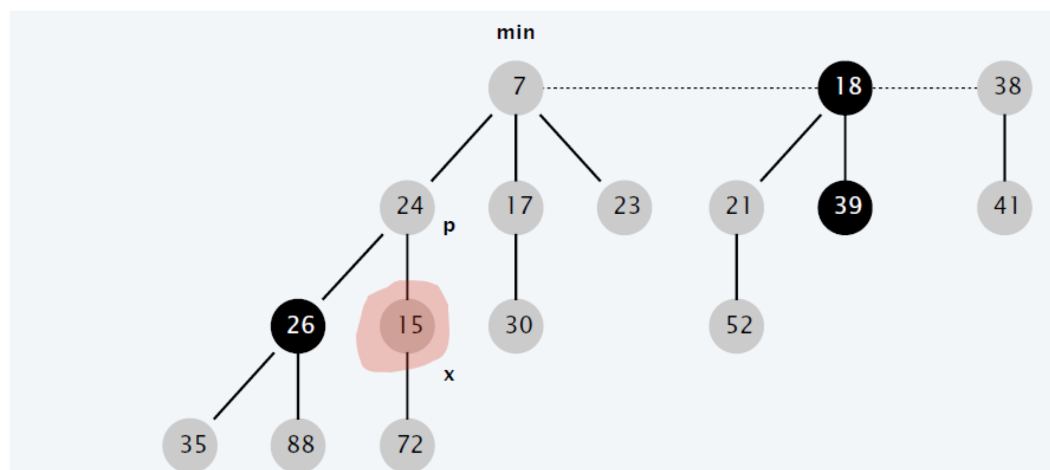
- Slučaj 1: Ovo je slučaj kada se poredak u okviru hipa ne menja, odnosno, *min* hip svojstva hipa ostaju zadovoljena. U okviru ovog slučaja, radi se smanjivanje vrednosti ključa, i nakon toga, ako je potrebno, promena *min* pokazivača

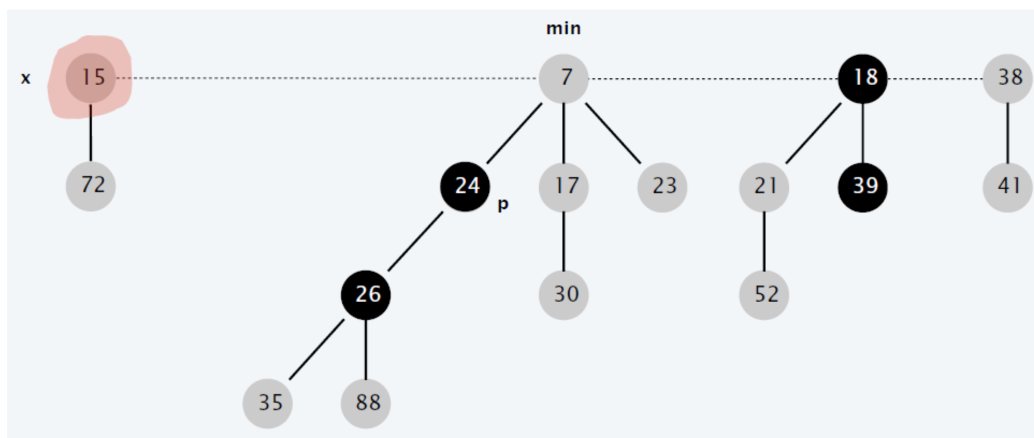




Slika 2.11 – Smanjenje vrednosti ključa Fibonačijevom hipu, slučaj 1.[20]

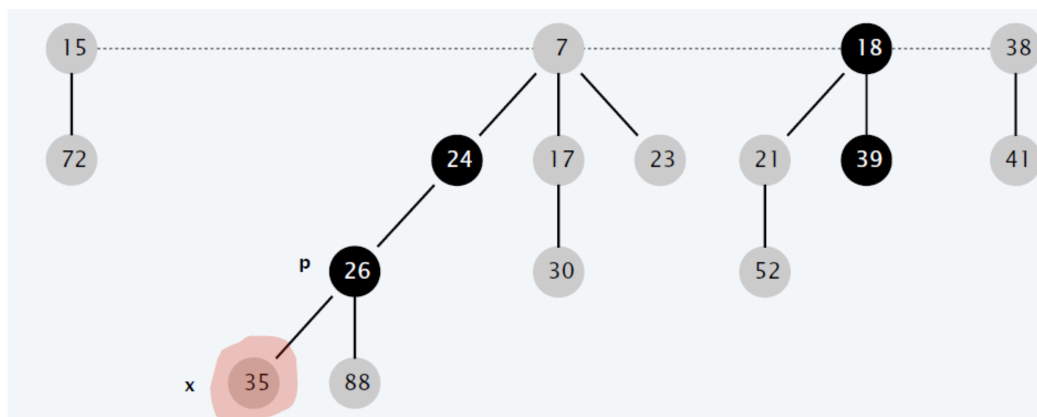
- Slučaj 2: Poredak u hipu se menja, čvor koji je promenio vrednost ključa, zajedno sa stablom kojem je roditelj se prebacuje u *root* listu. U zavisnosti da li je atribut *mark* kod roditelja tog čvora *true* ili *false*, može da se desi jedna od dve situacije:
  - ako roditelj nije markiran, odnosno, vrednost *mark* atributa je *false*, ta vrednost se postavlja na *true* (izuzetak je ako se roditelj nalazi u *root* listi, tada se ne setuje *mark* atribut)

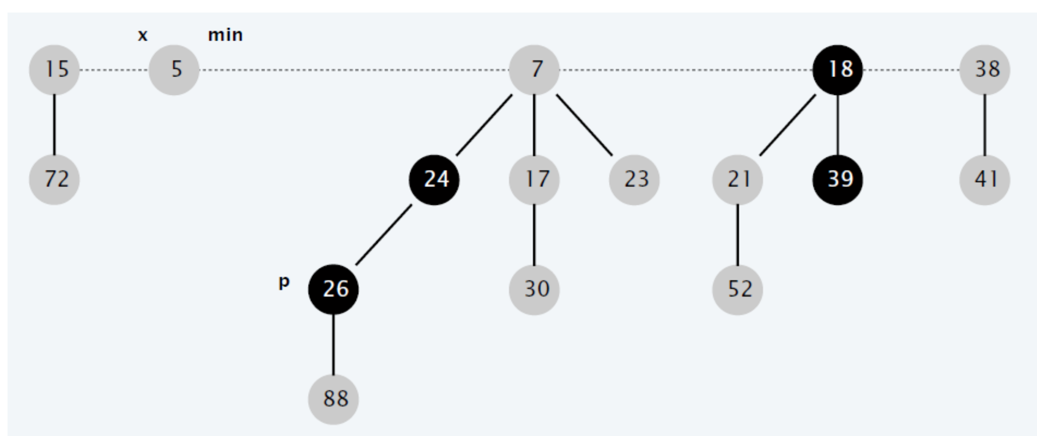


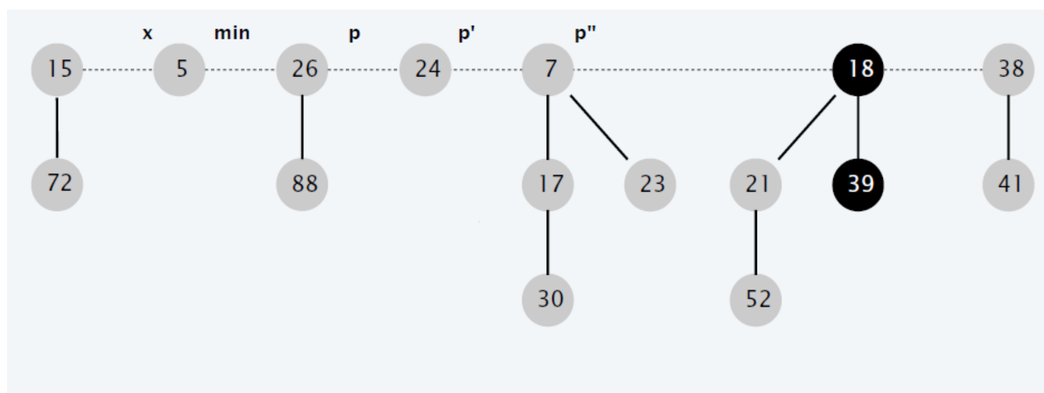


Slika 2.12 – Smanjenje vrednosti ključa Fibonačijevom hipu, slučaj 2, prva situacija [20]

- ako je roditelj markiran, roditelj se takođe prebacuje u *root* listu, i mark atribut mu se setuje sa *true* na *false*. Ova provera se ponavlja sve do vrha stabla u kojem se čvor nalazi, sve dok se ne naiđe na čvor čija je vrednost *mark* atributa *false*







Slika 2.13 – Smanjenje vrednosti ključa Fibonačijevom hipu, slučaj 2, druga situacija [20]

Svaki od ovih slučajeva se odvija u *O(1)* vremenskoj složenosti, tako da je i ukupna vremenska složenost ove operacije *O(1)*.

Brisanje čvora

Ova operacija se realizuje na isti način kao i kod binomijalnog hipa, odnosno sastoji iz dva dela: umanjenja vrednosti ključa koji se briše na minimalnu vrednost, tako da taj čvor sigurno postane minimalni čvor u celom hipu, kao i operacije brisanja čvora sa najmanjim ključem.

Operacija brisanje čvora sa najmanjom vrednosti ključa ima vremensku složenost *O(log n),* a smanjivanje vrednost ključa *O (1),* tako da je vremenska složenost ove operacije *O(log n).*

2.4. Korišćenje hipova

Upotreba hipova je vrlo raznovrsna. Najčešće se hip koristi za implementaciju prioritetnog reda. Prioritetni red je apstraktna struktura podataka slična steku ili redu, ali sa dodatkom da svaki element ima svoj prioritet. U okviru ove strukture podataka, element sa "višim" prioritetom ima prednost u odnosu na element sa "nižim" prioritetom. Ako dva elementa imaju isti prioritet, pre će biti opslužen element koji se po redosledu nalazi ispred. Sam pojam prioritetni red, često se menja sa pojmom hip, međutim, hip je samo jedan od načina implementacije prioritetnih redova. Pored hipa, prioritetni redovi mogu biti implementirani na različite drugo načine, kao što su niz, ulančana lista, itd.

Hip se dosta koristi u mnogim algoritmima kao pomoćna struktura. Tako je, na primer, jedan je od najpoznatijih i najefikasnijih algoritama sortiranja iz grupe 'metode selekcije' [4], *heapsort*, upravo i dobio ime po ovoj strukturi. S obzirom da efikasnost korišćenja hipa često dominantno određuje vreme izvršavanja nekih algoritama, u evaluacionoj analizi će biti ispitan uticaj vrste korišćenog hipa na Dijkstra-in i Huffman-ov algoritam, a u nastavku je dat vrlo kratak opis ova dva algoritma.

Dijkstra algoritam za nalaženje najkraćih rastojanja

U teoriji grafova, problem nalaženja najkraćeg rastojanja je problem pronalaženja puta između dva čvora u grafu tako da je zbir težina između grana koje ih povezuju najmanji. Najčešća upotreba ovog algoritma je pronalaženje puta između dve fizičke lokacije, kao npr. u okviru aplikacija koje se bave planiranjem ruta za putovanje. Pored ove upotrebe, ovaj algoritam se može naći i u nekim mrežnim ili telekomunikacionim algoritmima, robotici ili VLSI dizajnu. Zanimljiva je i primena ovog algoritma za rešavanje problema Rubikove kocke, gde svaka usmerena grana odgovora jednom potezu, a algoritam najkraćeg puta se koristi za nalaženje rešenja sa najmanjim brojem poteza.

Osnovni delovi Dijkstra algoritma za nalaženje najkraćih rastojanja, pri implementaciji preko hipova su:

- ubacivanje svih čvorova iz grafa u hip, tako da je put, odnosno ključ za početni čvor 0, a za ostale čvorove beskonačno
- uzimanje najmanje veličine puta iz hipa, odnosno uzimanje čvora sa najmanjim ključem i proverava za sve njegove susede iz hipa, da li je dotadašnja vrednost puta do tih suseda manja od vrednosti puta koja se dobija sabiranjem vrednosti puta uzetog čvora i vrednosti puta od uzetog čvora do tog suseda. Ako je dotadašnja vrednost veća, radi se umanjenje vrednosti puta trenutno uzetog suseda
- prethodni korak se ponavlja dok se ne prođu svi čvorovi

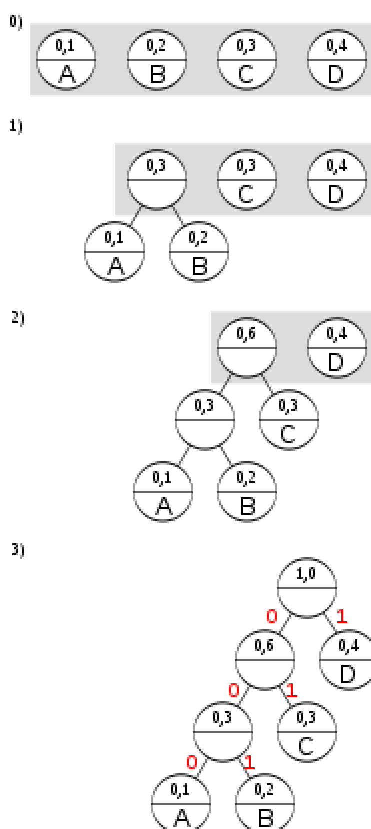
Kao što se može videti, osnovno operacije su ubacivanje novog elementa, brisanje elementa sa najmanjom vrednosti ključa i umanjenje vrednosti ključa. Vremenska složenost ovog algoritma, pri implementaciji sa binarnim hipom je $O(|E| + |V|log\frac{|E|}{|V|}| \ log \ |V|)$[9], a pri implementaciji sa Fibonačijevim hipom je $O(|E| + |V| \ log \ |V|)$, gde $|E|$ predstavlja broj čvorova u grafu, a $|V|$ broj grana.

Hafmanovi kodovi

U oblasti komunikacije i računarske tehnike, Hafmanovo kodiranje predstavlja algoritam za kodiranje simbola bez gubitka informacija. Ovaj algoritam kreće od šume stabala sa po jednim čvorom. U svakoj iteraciji, iz reda u kome se nalaze stabla, vade se dva korena sa najmanjim težinama i zatim se pravi čvor sa težinom koja je jednaka zbiru težina izabranih čvorova. Sada je novonastali čvor koren novog stabla čije su levo i desno podstablo izabrana dva čvora. Nakon toga, novonastalo stablo se vraća u red svih čvorova i u narednim iteracijama ravnopravno konkuriše sa ostalim stablima. Kao što se može videti, osnovne dve operacije su uzimanje čvora sa najmanjom

vrednošću, odnosno umetanje novog čvora, tako da hip predstavlja idealnu strukturu pri radu sa ovim algoritmom.

Hafmanovi kodovi najčešće se koriste za kompresiji podataka pri njihovom čuvanju ili prenosu da bi se smanjilo memorijsko zauzeće ili vreme prenosa. Jedan od primera su poruke koje čine azbučna slova. Prilikom kodovanja poruke mora se voditi računa da procesi kodovanja i dekodovanje budu jednostavni, a opet dužina poruke treba da bude minimalna. Svaki simbol se predstavlja nekim binarnim kodom i za svaki simbol se mora odrediti verovatnoća pojavljavljivanja, tako da zbir svih verovatnoća bude 100 [4]. Zbog lakog dekodovanja, koriste se prefiksni kodovi, što znači da kod ni jednog simbola ne sme da bude početak drugog.



Slika 2.14 – Primer rada Hafmanovog algoritma, odnosno Hafmanovih kodova [21]

3. IMPLEMENTACIJA HIPOVA

U ovom poglavlju će biti objašnjene i prikazane implementacije hipova, kroz implementacije operacija.

3.1. Umetanje elementa

Binarni hip

Kod binarnog hipa, osnovni deo algoritma se sastoji od postavljanja novog elementa sa ključem na trenutno poslednje mesto u hipu*(1)* i metodom "zamenom na gore" (*shiftUp*), koja dati čvor menja sa čvorem roditelja *(2).*

```
void insert (int key)
{
      heap[heapCurrSize++] = key; //(1)
      shiftUp(heapCurrSize - 1); //(2)
}
```

Metoda *shiftUp* pomera čvor ka vrhu hipa sve dok ključ trenutnog čvora ima veću vrednost od čvora roditelja. Kada se desi situacija da čvor roditelja ima manji ključ od deteta koji se gleda, metoda se završava *(1).* Funckija *getParentIndex(index)*, za tekući čvor odnosno indeks, pronalazi indeks čvora roditelja *(2).* Ovu metodu je moguće realizovati rekurzivno ili iterativno, a u nastavku je dat prilog za iterativnu realizaciju.

```
void shiftUp (int index)
{
      if (index != 0)
      {
            int parentIndex = getParentIndex(index); //(2)
```

```
            int tmp;
            while (heap[parentIndex] > heap[index]) //(1)
            {
                  tmp = heap[parentIndex];
                  heap[parentIndex] = heap[index];
                  heap[index] = tmp;
                  index = parentIndex;
                  parentIndex = getParentIndex(index); //(2)
            }
      }
}}
```

Binomijalni hip

Kod binomijalnog hipa, operacija umetanja novog elementa se obavlja kreiranjem novog binomijalnog hipa *(1),* setovanjem *head* pokazivača(pokazivač koji pokazuje na prvi element u *root* listi) tek kreiranog hipa na elementa koji se umeće *(2),* i pozivanja operacije unije hipova, koja će u hip koji se koristi dodati novi hip *(3).* Implementacija operacije unije dva hipa će biti prikazana u nastavku rada.

```
void insert (Node* newNode)
{
      BinomialHeapBasic* bh = new BinomialHeapBasic();//(1)
      bh->head = newNode; //(2)
      unionHeaps(bh); //(3)
}
```

Fibonačijev hip

U okviru ove vrste hipa, nakon setovanja određenih pokazivača i vrednosti na čvor koji se ubacuje *(1),* radi se umetanje novog čvora u *root* (dvostruko ulančanu sortiranu) listu. U zavisnosti da li je hip u koji se ubacuje novi čvor prazan ili ne *(2),* izvršava se taj proces i setuje se pokazivač na najmanju vrednost ključa u hipa - *min (3).* Na kraju operacije, brojač za ukupan broj čvorova u hipu se povećava *(4).*

```
void insert (Node* newNode)
{
      newNode->degree = 0; //(1)
      newNode->mark = false; //(1)
      newNode->child = nullptr; //(1)
      newNode->parent = nullptr; //(1)

      if (min == nullptr) //(2)
      {
            min = newNode; //(3)
            min->left = min;
            min->right = min;
      }
      else  //(2)
      {
            insertIntoDoubleLinkedtList(head, newNode);
            if (newNode->key < head->key)
            {
```

```
                min = newNode; //(3)
            }
        }

        numberNodes++; //(4)
}
```

3.2. Nalaženje elementa sa najmanjim ključem

Binarni hip

Čvor sa najmanjim ključem se nalazi na početku hipa, tako da se veoma jednostavno dobija ta vrednost.

```
int minimum ()
{
        return heap[0];
}
```

Binomijalni hip

Kod binomijalnog hipa, proces je malo složeniji, odnosno prolazi se kroz *root* (jednostuko ulančanu) listu i traži se čvor čija je vrednost ključa najmanja.

```
int minimum ()
{
        int min = head->key;
        Node* iterator = head->sibling;
        while (iterator != nullptr)
        {
                if (iterator->key < min)
                {
                        min = iterator->key;
                }
                iterator = iterator->sibling;
        }
        return min;
}
```

Fibonačijev hip

Na čvor sa najmanjim ključem, u okviru Fibonačijevog hipa pokazuje pokazivač *min*. Izvlačenje ključa sa tog pokazivača dobija se i najmanja vrednost ključa na nivou celog hipa.

```
int minimum ()
{
        return min->key;
}
```

3.3. Brisanje elementa sa najmanjim ključem

Binarni hip

U okviru ove vrste hipa, element sa najmanjom vrednošću ključa nalazi se u korenu hipa *(1).* Nakon uzimanje te vrednosti, sam proces uklanjanja elementa sa tim ključem se sastoji iz smanjivanja vrednosti ukupnog broja elemenata u hipu *(2),* stavljanja poslednjeg elementa u hipu na koren *(3),* i metode shiftDown *(4)*

```
int extractMinimum ()
{
      int minimumElement = heap[0]; //(1)
      heapCurrSize--;               //(2)
      heap[0] = heap[heapCurrSize]; //(3)
      if (heapCurrSize > 0)
      {
            shiftDown(0);            //(4)
      }
      return minimumElement;
}
```

Metoda *shiftDown* pomera element sa indeksom koji se nalazi kao parametar te metode ka listovima, na taj način što za trenutno uzeti element, traži dete koji ima manju vrednost ključa, i upoređuje sa ključem uzetog elementa. Ako je ključ trenutnog elementa veći od ključa nađenog deteta, radi se zamena ta dva elementa i proces se nastavlja, a ako nije, ova metoda se zavšava. Ova metoda može da se realizuje rekurzivno i iterativno, a u ovom radu dat je prikaz sa iterativnom realizacijom. Metode *getLeftSonIndex* i *getRightSonIndex* nalaze indekse levog, odnosno desnog deteta.

```
void shiftDown(int index)
{
      int currIndex = index;
      int leftSonIndex = getLeftSonIndex(currIndex);
      while(leftSonIndex < heapCurrSize)
      {
            if (leftSonIndex >= heapCurrSize)
                  return;
            int rightSonIndex = getRightSonIndex(currIndex);

            //odredjivanje koji je indeks min
            int minSonIndex;
            if (rightSonIndex >= heapCurrSize)
            {
                  minSonIndex = leftSonIndex;
            }
            else  //ako postoje oba deteta, koji je manji izmedju njih
            {
                  if (heap[leftSonIndex] <= heap[rightSonIndex])
                  {
```

```
                        minSonIndex = leftSonIndex;
                }
                else
                {
                        minSonIndex = rightSonIndex;
                }
        }

        if (heap[currIndex] > heap[minSonIndex])
        {
                std::swap(heap[currIndex], heap[minSonIndex]);
        }

        currIndex = minSonIndex;
        leftSonIndex = getLeftSonIndex(minSonIndex);
    }
}
```

Binomijalni hip

Prvi deo operacije brisanja čvora sa najmanjim ključem kod binomijalnog hipa se sastoji od pronalaženja čvora sa najmanjim ključem, čija implementacija je već prikazana u prethodnom delu. Jedini dodatak na taj deo je i čuvanje informacije o prethodnom čvoru, s obzirom da je ona potrebna kako bi se jednostavnije odradilo brisanju tekućeg čvora u jednostruko ulančanoj listi.

```
    Node* iterator = head;
    Node* nodewWithMin = iterator;
    Node* prevNodeWithMin = nullptr;
    while (iterator->sibling != nullptr)
    {
        if (iterator->sibling->key < nodewWithMin->key)
        {
            prevNodeWithMin = iterator;
            nodewWithMin = iterator->sibling;
        }
        iterator = iterator->sibling;
    }
```

Nakon pronalaženja čvora sa najmanjim ključem, radi se uklanjanje tog čvora, uz proveru da ako pokazivač *head*, odnosno pokazivač na *root* listu pokazuje i na čvor koji se briše, dolazi do pomeranja tog pokazivača (1). U suprotnom, radi se uklanjanje elementa iz jednostruko ulančane liste.

```
    Node* iterator = head;
    if (prevNodeWithMin == nullptr)
    {
        head = nodewWithMin->sibling; //(1)
    }
    else
    {
        prevNodeWithMin->sibling = nodewWithMin->sibling; //2)
    }
    nodewWithMin->sibling = nullptr;
```

Sledeći deo ove operacije se sastoji od obrtanja redosleda dece čvora sa minimalnom vrednošću ključa*(1)* i stavljanja tih čvorova u novoformirani hip *(2).* Poslednji korak je unija tog novoformiranog hipa sa tekućim hipom *(3).* Proces unije dva hipa će biti prikazan u nastavku rada.

```
BinomialHeapBasic* tmpBh = new BinomialHeapBasic();

Node* iterator = head;
if (nodewWithMin->child != nullptr)
{
      Node* prevNode = nullptr;
      Node* currNode = nodewWithMin->child;
      Node* nextNode = currNode->sibling;
      currNode->parent = nullptr;
      while (nextNode != nullptr) //(1)
      {
            currNode->sibling = prevNode;
            prevNode = currNode;

            currNode = nextNode;
            nextNode = nextNode->sibling;

            currNode->parent = nullptr;
      }
      currNode->sibling = prevNode;
      tmpBh->head = currNode; //(2)
}

unionHeaps(tmpBh);    //(3)
```

Fibonačijev hip

Kod ove vrste hipa, prvi deo operacije brisanja čvora sa najmanjim ključem je prebacivanja dece od čvora sa najmanjim ključem u *root* listu *(1).* Nakon toga, ako u tom hipu postoji samo jedan čvor, on se ujedno i briše i pokazivač *min* na najmanji čvor u hipu se setuje na *null* vrednost *(2)*, dok u suprotnom dolazi do brisanja čvora sa minimalnom vrednošću hipa i poziva metode *consolidate (3).* Poslednji deo je smanjivanje ukupnog broja čvorova u hipu *(4).*

```
-Prebacivanje dece čvora sa najmanjim ključem u root listu-  (1)

if (minNode == minNode->left)
{
      min = nullptr; (2)
}
else
{
      -brisanje čvora sa minimalnom vrednošću ključa-

      consolidate(); //(3)
}

numberNodes--; //(4)
```

Metoda *consolidate()* prolazi kroz sve čvorove iz *root* liste, tako da je cilj nakon završetka ove metode da u root listi ne postoje dva čvora sa istim stepenom. Na početku ove metode uvodi se

nova promenljiva, odnosno niz *rank* u kojem će se nalaziti čvorovi iz root liste, tako da nulta pozicija u nizu pokazuje na čvor sa stepenom nula, prva pozicija na čvor sa stepenom jedan itd. Veličina ovog niza je *log(n). (1)*

Sledeći korak je prolazak kroz sve čvorove iz *root* liste *(2)* i provera da li već postoji neki upisan čvor u *rank* nizu sa tom vrednošću stepena *(3).* U slučaju da ne postoji, radi se ubacivanje tog čvora u *rank* niz *(4),* dok u suprotnom poziva se procedure *fibHeapLink* koja spaja ta dva čvora *(5),* setuje trenutnu poziciju za ova dva čvora u *rank* nizu na *null*(6) i povećava stepen novoformiranog čvora za jedan *(7).* Ovo spajanje se obavlja u petlji, jer može da se desi da nakon spajanja, taj novi čvor se ponovo pokuša sa ubacivanjem u *rank* niz, i da je i njegovo mesto zauzeto. Petlja se izvršava sve dok se ne dođe do mesta u *rank* nizu koje je slobodno.

Poslednji korak je pravljenje nove *root* liste od čvorova iz rank liste *(8).*

```
void consolidate()
{
      int upperBound = log(numberNodes); //(1)
      Node** rank = new Node*[upperBound];

      while(...)// prolazak kroz sve čvorove u root listi  // (2)
      {
            currentNodeDegree = currentNode->degree; //stepen čvora

            while(consArray[currentNodeDegree] != nullptr) //(3)
            {
                  Node* nodeWithSameDegree = rank [currentNodeDegree];
                  if(currentNode->key > nodeWithSameDegree->key)
                        swap(currentNode, nodeWithSameDegree);
                  fibHeapLink(nodeWithSameDegree, currentNode); //(5)
                  rank [currentNodeDegree] = nullptr; //(6)
                  currentNodeDegree++; //(7)
            }
           rank [currentNodeDegree] = currentNode;//(4)
           currentNode = sledeći čvor iz root liste;
      }
      -pravljenje nove root liste na osnovu rank liste-  // (8)
}
```

3.4. Smanjivanje vrednosti ključa

Prvi korak prilikom izvršenja ove operacije, kod svih vrsta hipova je provera da li je vrednost novog ključa manja ili jednaka vrednosti starog ključa. Ako jeste, operacija se nastavlja u zavisnosti od vrste izabranog hipa.

Binarni hip

Operacija umanjenja vrednost ključa kod binarnog hipa se sastoji od same promene vrednosti ključa izabranog elementa i metode *shiftUp* koja je opisana u operaciji umetanja novog elementa.

```
void decreaseKey(int index, int value)
{
	heap[index] = value;

	shiftUp(index);
}
```

Binomijalni hip

Slično kao kod binarnog hipa, prvi deo se sastoji od same promene vrednosti ključa izabranog čvora, dok drugi deo predstavlja pomeranje čvora sa tim ključem ka vrhu hipu. To pomeranje se obavlja pomoću metode *swap* koja menja dva čvora u binomijalnom hipu, a pre toga proverom da li čvor roditelja ima manju vrednost ključa od čvora koji se u tom trenutku gleda. Provera se obavlja sve dok se ne dođe do vrha hipa, odnosno čvora sa manjom vrednošću ključa od trenutnog čvora.

```
void decreaseKey(Node* currNode, int value)
{
	currNode->key = newKey;

	Node* parentNode = currNode ->parent;

	while (parentNode != nullptr && currNode->key < parentNode->key)
	{
		swap(currNode ->key, parentNode->key);
		currNode = parentNode;
		parentNode = parentNode->parent;
	}
}
```

Fibonačijev hip

Slično kao kod binarnog hipa, prvi deo se sastoji od same promene vrednosti ključa izabranog čvora, dok drugi deo predstavlja pomeranje čvora sa tim ključem ka vrhu hipa. To pomeranje se obavlja pomoću metode *swap* koja menja dva čvora u binomijalnom hipu, a pre toga proverom da li čvor roditelja ima manju vrednost ključa od čvora koji se u tom trenutku gleda. Provera se obavlja sve dok se ne dođe do vrha hipa, odnosno čvora sa manjom vrednošću ključa od trenutnog čvora.

```
void decreaseKey(Node* currNode, int value)
{
	currNode->key = newKey;

	Node* parentNode = currNode ->parent;

	while (parentNode != nullptr && currNode->key < parentNode->key)
	{
		swap(currNode ->key, parentNode->key);
		currNode = parentNode;
		parentNode = parentNode->parent;
	}
}
```

3.5. Brisanje elementa

Binarni hip

Operacija brisanja elementa kod binarnog hipa se sastoji od zamene čvora koja si briše sa poslednjim elementom u hipu, smanjivanja vrednosti ukupnog broja čvorova u hipu i pozivanjem metoda *shiftUp*, odnosno *shiftDown* u zavisnosti od vrednosti ključa čvora koji je prebačen sa poslednjeg mesta u hipu, na mesto onog koji se briše. Ako je ključ čvora koji je prebačen manji od vrednosti ključa koji se briše, radi se operacija *shiftUp*, u suprotnom *shiftDown*.

```
void delete(int index)
{
      if (index >= 0 && index < heapCurrSize)
      {
            heap[index] = heap[--heapCurrSize];

            if (heap[index] < heap[heapCurrSize])
            {
                  shiftUp(index);
            }
            else
            {
                  shiftDown(index);
            }
      }
}
```

Binomijalni hip

Kod binomijalnog hipa, operacija brisanja čvora se sastoji od poziva dve već obrađene operacija – umanjenja vrednosti ključa čvora na najmanju moguću vrednost i brisanja čvora sa najmanjom vrednošću.

```
void delete(Node* currNode)
{
      decreaseKey(currNode, -1);
      exctractMinimum();
}
```

Fibonačijev hip

Operacija brisanja čvora kod Fibonačijevog hipa se obavlja na isti način kao i kod binomijalniog.

```
void delete(Node* currNode)
{
      decreaseKey(currNode, -1);
      exctractMinimum();
}
```

3.6. Unija dva hipa

Binarni hip

Prvi korak kod operacije unije dva hipa kod binarnog hipa je prekopiranje hipa koji se ubacuje u trenutni hip *(1)*. Nakon toga izvršava se metoda *createHeapFromArray(),* koja kreira novi hip koji zadovoljava, u ovom slučaju, *min-heap* uslove.

```
void union(BinaryHeap* newHeap)
{
      for (int i = 0; i < newHeap->getHeapCurrSize(); i++) //(1)
      {
            heap[heapCurrSize++] = heap1->getHeap()[i];
      }
      createHeapFromArray();
}
```

Metoda *createHeapFromArray()* prolazi kroz sve elemente hipa, počevši od poslednjeg elementa, i radi proveru ključa tog elementa sa ključem elementa roditelja. Ako je ključ elementa koji se gleda manji od ključa roditelja, radi se zamena. Nakon završetka ove metode, hip će imati *min* hip osobine.

```
void createHeapFromArray()
{
      for (int index = getHeapCurrSize() - 1; index > 0; index--)
      {
            int parentIndex = getParentIndex(index);
            if (getElementByIndex(index)<getElementByIndex(parentIndex))
            {
                  swapElementsByIndex(index, parentIndex);
            }
      }
}
```

Binomijalni hip

Operacije unije kod binomijalnog hipa je osnovna operacije, s obzirom da se koristi u okviru svih ostalih operacija. Prvi korak je prekopiranje *root* liste iz hipa koji se ubacuje u trenutni hip, koje se obavlja pomoću metode *binomialHeapHeadMerge(1).* Nakon toga, radi se prolazak kroz sve čvorove u *root* listi *(2)* i u zavisnosti od situacije, radi se spajanje čvorova, ili izvršavanje neke od metoda.

Osnovna situacija pri prolasku kroz sve čvorove u *root* listi je da ako dva susedna čvora imaju različite stepene ili tri susedna čvora imaju isti stepen, radi se prelazak na sledeći čvor*(3).* U suprotnom, ako dva susedna čvora imaju isti stepen, a trenutni ima manju ili jednaku vrednost ključa kao sledeći, radi se spajanje ta dva čvora, tako da sledeci čvor postaje dete od trenutnog *(4).* Ako sledeći čvor ima manju vrednost od trenutnog, radi se isto povezivanje, samo što u ovoj situaciji trenutni čvor postaje dete od sledećeg čvora *(5).*

```
void union(BinomialHeap* newHeap)
{
	head = binomialHeapHeadMerge(newHeap); //(1)

	if (head == nullptr)
	{
		return;
	}

	Node* prevNode = nullptr;
	Node* currNode = head;
	Node* nextNode = currNode->sibling;

	while (nextNode != nullptr) //(2)
	{
		if ((currNode->degree != nextNode->degree) ||
			(nextNode->sibling != nullptr && nextNode->sibling
			  ->degree == currNode->degree)) //(3)
		{
			prevNode = currNode;
			currNode = nextNode;
		}
		else
		{
			if (currNode->key <= nextNode->key) //(4)
			{
				currNode->sibling = nextNode->sibling;
				binomialNodeLink(currNode, nextNode);
			}
			else //(5)
			{
				if (prevNode == nullptr)
				{
					head = nextNode;
				}
				else
				{
					prevNode->sibling = nextNode;
				}
				binomialNodeLink(nextNode, currNode);
				currNode = nextNode;
			}
		}
		nextNode = currNode->sibling;
	}}
```

Metoda *binomialNodeLink* radi spajanje dva čvora tako da čvor koji je prvi parametar postaje roditelj čvora koji je drugi parametar.

```
void binomialNodeLink (Node* parentNode, Node* childNode)
{
	childNode->parent = parentNode;
	childNode->sibling = parentNode->child;
	parentNode->child = childNode;
	parentNode->degree++;
}
```

Fibonačijev hip

Operacija unije dva hipa kod Fibonačijevog hipa je jednostavna i sastoji se od spajanja dve dvostruko kružne ulančane liste *(1).* Nakon spajanja, radi se setovanje pokazivača min u zavisnosti da li je vrednost ključa tog pokazivača kod hipa koji je ubačen manja od te vrednosti dotadašnjeg trenutnog hipa *(2)*, kao i setovanje ukupnog broj čvorova u novom hipu *(3).*

```
void union(FibonacciHeap* newHeap)
{
	if(min != nullptr && newHeap->min != nullptr) //(1)
	{
		//prevezivanje krajeva dve kruzne liste
		min->right->left = newHeap->min->left;
		newHeap->min->left->right = min->right;

		//spajanja min dve kruzne liste
		min->right = newHeap->min;
		newHeap->min->left = min;
	}
	else if (newHeap->min!= nullptr)
	{
		min = newHeap->min;
	}
	else
	{
		return;
	}

	if (newHeap->min!= nullptr && newHeap->minimum() <
minimum())//(2)
	{
		head = newHeap->min;
	}

	numberNodes += newHeap->numberNodes; //(3)
}
```

3.7. Implementaciona iskustva

Tokom procesa implementacije, skoro svaka operacija je imala neke svoje izazove koje je trebalo rešiti na način koji će minimalno uticati na brzinu izvršavanja te operacija. Kod Fibonačijevog hipa, kod operacija umetanja novog elementa, postojalo je nekoliko različitih načina umetanja novog čvora u dvostruko ulančanu sortiranu listu (metoda *insertIntoDoubleLinkedtList*). Takođe kod Fibonačijevog hipa, implementacija metode *consolidate* je imala dosta izazova, od smeštanja i obrade svih čvorova iz *root* liste u *rank* niz, i nakon toga re-kreiranja *root* liste.

Kod binarnog hipa, nije bilo puno izazova, i jedina nedoumica je bila način implementacija metoda *shiftDown*, odnosno *shiftUp*, odnosno da li će to biti iterativno ili rekurzivno. Očekivano, kod binomijalnog hipa operacija unije dva hipa, koja se koristi kod skoro svih ostalih operacija, je zahtevala pažljivu implementaciju i dosta testiranja.

4. METODOLOGIJA ANALIZE

U ovom poglavlju je opisana metodologija analize operacija sa hipovima. Do detalja je opisano test okruženje i navedena kako razvojna, tako i hardverska platforma na kojoj je vršena evaluacija algoritama.

4.1. Razvojna platforma

Za implementaciju operacija za rad sa hipovima korišćen je programski jezik C++. Od prevodilaca, prilikom prevođanja i testiranja, korišćen je *Microsoft Visual Studio.*

Microsoft Visual Studio je jedno od najpoznatijih integrisanih razvojnih okruženja (IDE). U svom sastavavu ima *code editor*, *debugger, designer,* kao i još neke dodatne alate. *Code editor* omnogućava isticanje sintakse(*syntax highlighting)* odnosno stavljanje u različite boje i fontove različitih tipova naredbi u okviru izvornog koda. Pored ove, postoji i mogućnost automatskog dovršavanja neke instrukcije (*autocomplete*). Za razvoj različitih vrsta aplikacija, *Microsoft Visual Studio* ima nekoliko proizvoda. U okviru ovog rada koristi se *Microsoft Visual C++,* a pored njega postoje *Microsoft Visual C#, Microsoft Visual Basic, Microsoft Visual Web Developer* i *Team Foundation Server. Microsoft Visual C++* je *Microsoft*-ova implementacija C i C++ prevodioca. Za programski jezik C, podržava ISO C standard sa delovima C99 specifikacije zajedno sa Microsoft-ovim specifičnim dodacima u obliku biblioteka. Što se tiče programskog jezika C++, podržava ANSI C++ specifikaciju zajedno C++11 dodacima [16]. U okviru ovog rada, na nekoliko mesta poput tajmera i generisanja različitih vrsta ulaznih raspodela je došlo do upotrebe C++11 dodataka.

Hardversku platformu za implementaciju i testiranje algoritama je činio računar zasnovan na *Intel I3* procesoru. Zasnovan je na mikroarhitekturi poznatoj pod imenom *Sandy Bridge*, a radni takt je *3.1 GHz* [17][16].

4.2. Test okruženje

Da bi bila sprovedena adekvatna analiza izabranih operacija i algoritama realizovano je test okruženje u kojem je moguće brzo i jednostavno podešavati vrednosti različitih parametara testiranja na osnovu kojih se može pokretati jedna od operacija ili grupa operacija prikazanih u ovom radu. U okviru samog pokretanja algoritama, omogućeno je merenje i ispisivanje vremena. Parametri koji se mogu podešavati u okviru test okruženja su:

- veličina hipa: omogućen je unos željene veličine hipa. U okviru ovog rada, prilikom analize osnovnih operacija, koristiće se ukupno pet različitih veličina nizova: 64K, 256K, 1M, 4M i 16M elemenata
- omogućen je izbor jedne od različitih vrsta ulaznih raspodela elemenata. U okviru ovog rada, omogućen je izbor između uniformne raspodele i raspodele gde elementi koji su se prethodno javljali imaju veću verovatnoću javljanja od ostalih elemenata. Kod uniformne raspodele elemenata, koriste se mogućnosti *C++11* standardna, odnosno konkretno biblioteka *uniform_int_distribution*.Što se tiče raspodele gde elementi koji su se prethodno javljali imaju veću verovatnoću javljanja nego ostali, ovaj način generisanja funkcioniše tako što element koji se poslednji javio ima verovatnoću javljanja pri sledećem generisanju 30%, prethodni koji se generisao 15%, element pre njega 7,5%, itd. za prethodnih deset elemenata. Svi ostali elementi imaju proporcionalnu verovatnoću javljanja.
- u zavisnosti od tipa operacije, kako bi se dobili konzistentni podaci, postavlja se i broj ponavljavanja izvšavanja te operacije. U okviru diskusije rezultata analize operacije, biće naglašeno koje su to operacije i koji je ukupan broj ponavljanja.

4.2.1. Merenje vremena

Merenje vremena je realizovano uz pomoć C++11 biblioteke *<chrono>*, odnosno klase *std::chrono::high_resolution_clock.* Ova klasa reprezentuje časovnik sa najmanjim otkucajnim periodom u okviru C++11 standarda. Vreme u datom trenutku se dobija pozivom funkcije *chrono::high_resolution_clock::now()* i pamti se u okviru promenljive iz iste biblioteke *chrono::high_resolution_clock::time_point.* Tačan vremenski interval između dva vremena se može odrediti pomoću prikazane komande *(1),* i nakon toga pozivom funkcion *count (2).*

```
chrono::duration<double> time_span= chrono::duration_cast
<chrono::duration<double>>(endTime - startTime);//(1)
time_span.count();//(2)
```

5. REZULTATI ANALIZE

U ovom poglavlju su izneseni relevantni rezultati i data je diskusija rezultata u odnosu na očekivane vrednosti i donesene projektne odluke. Poglavlje je podeljeno na dve celine: u okviru prve obrađena je analiza izabranih operacija, dok su u drugoj obrađene simulacije algoritama koje koriste hipove kao pomoćnu strukturu.

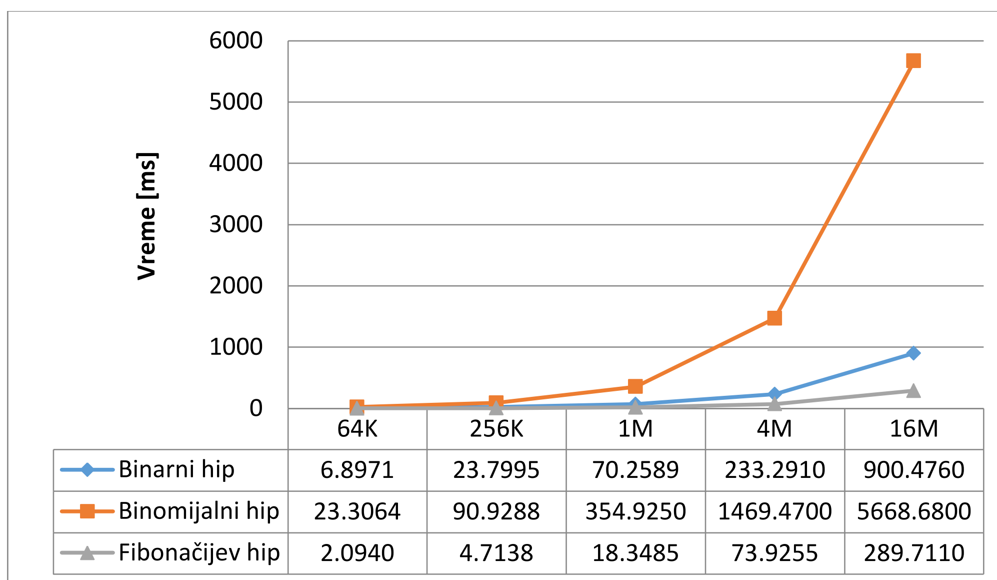
5.1. Osnovne operacije

U okviru ovog dela, za sve izabrane operacije biće prikazane zavisnosti vremena izvršavanja u odnosu na veličinu niza, sa uniformnom raspodelom elemenata. Pored prikaza tabela, nalazi se osnovna analiza rezultata za svaku operaciju, dok za neke operacije, odnosno tipove hipa, biće prikazana i dodatna analiza. Poslednji deo za svaku operaciju obuhvatiće tabelu i analizu rezultata po ulaznim raspodela za dve izabrane ulazne raspodele – uniformnu i raspodelu gde elementi koji su se poslednji javljali imaju veću verovatnoću javljanja u narednoj iteraciji.

5.1.1. Umetanje elementa

Gledajući vremenske složenosti ove opercije kod tri obrađene vrste hipa, očekivano bi bilo da Fibonačijev hip ima najbolje vreme izvršavanje, zato što je jedino kod njega vremenska složenost *O(1)*. Ovaj zaključak se i pokazao tačnim što se može videti na slici 5.1 međutim, zanimljivo je i poređenje umetanja novog elementa kod binarnog i binomijalnog hipa. Iako ova operacija i kod binarnog i kod binomijalnog hipa ima vremensku složenost *O(log n),* prilikom praktičnog rada i merenja binarni hip se pokazao mnogo povoljniji za rad. Dva osnovna razloga za takve rezultate su:

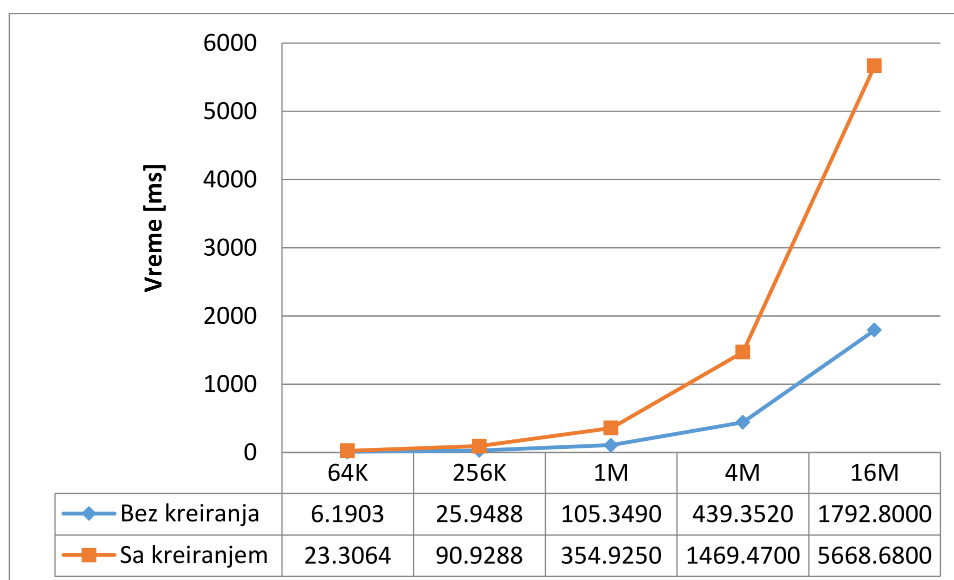
- prilikom umetanja novog elementa kod binomijalnog hipa, javlja se dodatni *overhead* koji se sastoji od kreiranja hipa za element koji se umeće
- sam način realizovanja ovih operacija, gde je kod binarnog hipa dosta jednostavno propagiranje elemenata od poslednjeg nivoa ka vrhu (sa malim brojem zamena odnosno instrukcija), za razliku od operacije unije dva hipa, koja se nalazi u osnovi operacije umetanja, i koja je dosta složenija.



	64K	256K	1M	4M	16M
Binarni hip	6.8971	23.7995	70.2589	233.2910	900.4760
Binomijalni hip	23.3064	90.9288	354.9250	1469.4700	5668.6800
Fibonačijev hip	2.0940	4.7138	18.3485	73.9255	289.7110

Slika 5.1 – Zavisnost vremena izvršavanja od ukupnog broja elementa kod operacije umetanja novog elementa

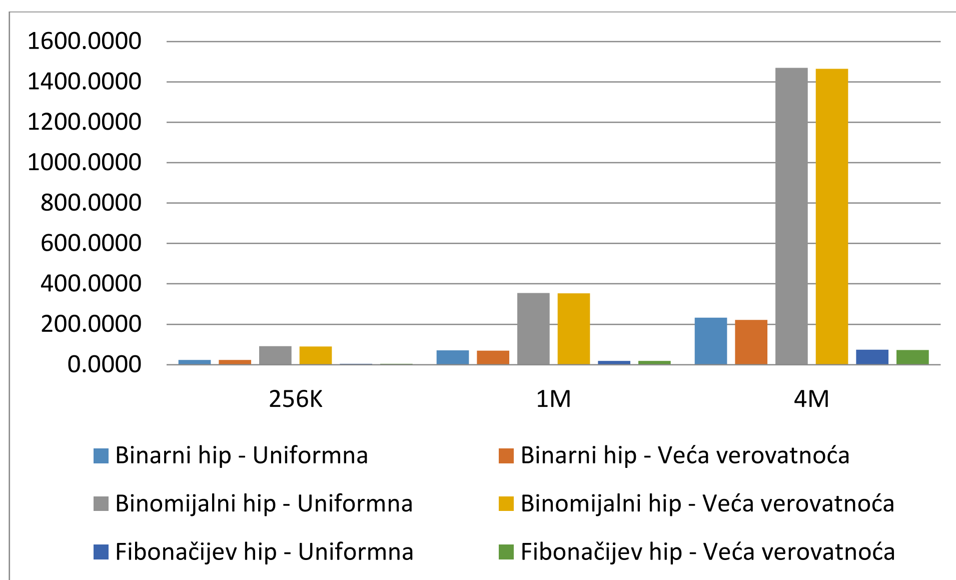
Slika 5.2 prikazuje uticaj kreiranja novog hipa na vreme izvršavanja, koji se javlja prilikom svakog umetanja novog elementa kod binomijalnog hipa. Ako u hipotetičkom slučaju, kreiranje novog hipa se ne bi obavljalo prilikom umetanja novog elementa, došlo bi do poboljšanja vremena ove operacije kod binomijalnog hipa.



	64K	256K	1M	4M	16M
Bez kreiranja	6.1903	25.9488	105.3490	439.3520	1792.8000
Sa kreiranjem	23.3064	90.9288	354.9250	1469.4700	5668.6800

Slika 5.2 - Umetanje novog elementa kod binomijalnog hipa, u slučaju kada se kreira, odnosno ne kreira novi hip

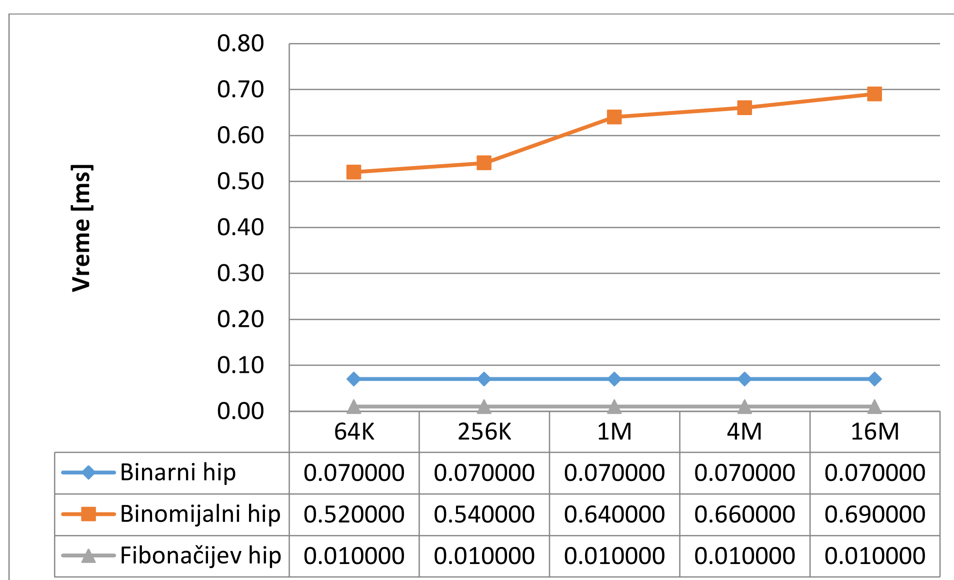
Izbor ulazne raspodele, za operaciju umetanja novog elementa nema veliki uticaj na brzinu ni kod jednog od izabranih hipova što prikazuje slika 5.3.



Slika 5.3 - Zavisnost vremena izvršavanja od ukupnog broja elementa pri poređenju različitih ulaznih raspodela kod operacije umetanja novog elementa

5.1.2. Nalaženje elementa sa najmanjim ključem

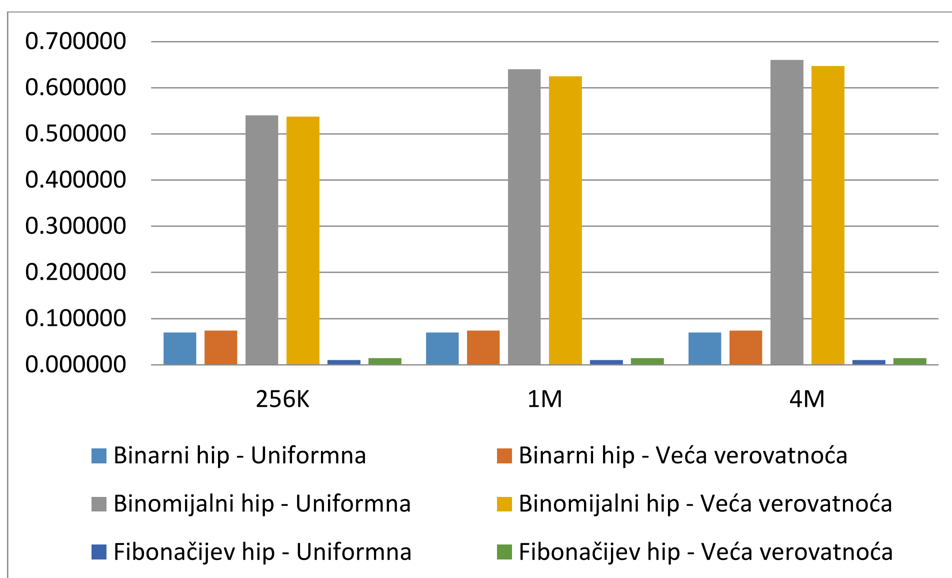
Ova operacija kod binarnog i Fibonačijevog hipa ima konstantnu vremensku složenost *O(1)*, dok je kod binomijalnog hipa logaritamska. Rezultati merenja sa slike 5.4 su to i potvrdili, odnosno ova operacija kod binomijalnog hipa je znatno sporija u odnosu na druge dve vrste testiranih hipova. Testiranje i merenje je rađeno sa 16K nalaženja čvora sa najmanjim ključem.



	64K	256K	1M	4M	16M
Binarni hip	0.070000	0.070000	0.070000	0.070000	0.070000
Binomijalni hip	0.520000	0.540000	0.640000	0.660000	0.690000
Fibonačijev hip	0.010000	0.010000	0.010000	0.010000	0.010000

Slika 5.4 - Zavisnost vremena izvršavanja od ukupnog broja elementa kod operacije nalaženja elementa sa najmanjim ključem

Kao i kod operacije umetanja novog elementa, izbor ulazne raspodele nema uticaj ne performanse pri operaciji nalaženja elementa sa najmanjim ključem što se i vidi na slici 5.5.



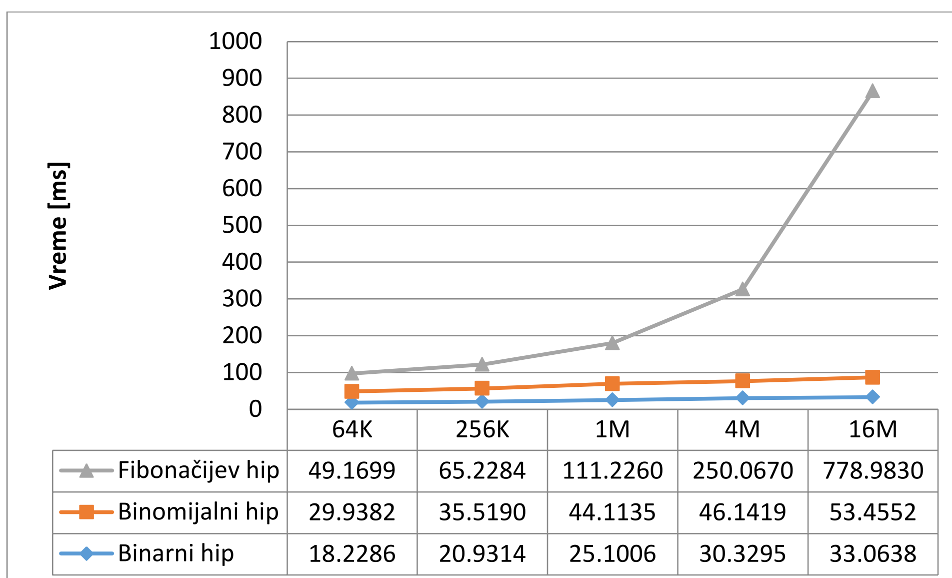
Slika 5.5 - Zavisnost vremena izvršavanja od ukupnog broja elementa pri poređenju različitih ulaznih raspodela kod operacije nalaženje elementa sa najmanjim ključem

Kod binomijalnog hipa može da se javi zanimljiva situacija. Kao što je objašnjeno u poglavlju 2.2, broj čvorova u root listi je jednak broju jedinica koje se nalaze u binarnom zapisu ukupnog broja čvorova u hipu. Ako npr. hip ima 4194304 elemenata, u *root* listi se nalazi samo jedan čvor, dok ako ima 4194303, u *root* listi se nalazi 22 čvora. Ova stavka može da ima mali uticaj na ovu operaciju, kao i operacije gde se radi pretraga odnosno prolazak kroz *root* listu.

5.1.3. Brisanje elementa sa najmanjim ključem

Brisanje elementa sa najmanjim ključem kod sve tri vrste obrađenih hipova ima vremensku složenost *O(log n)*. U praktičnom radu, tj. prilikom merenja koje prikazuje slika 5.6, najbolje vremenske performanse je ova operacija imala kod binarnog hipa. Glavni razlog je sama jednostavnost ove operacije kod binarnog hipa, odnosno, dosta jednostavna provera i zamena čvorova na dole. Kod binomijalnog hipa, algoritam za ovu operaciju sastoji se iz nekoliko delova sa vremenskim složenostima *O(log n)*. Dodatno usporenje pri praktičnom radu se javlja i prilikom kreiranja novog binomijalnog hipa prilikom svakog brisanja čvora sa najmanjim ključem, međutim i pored toga, ova operacija kod binomijalnog hipa daje dosta bolje performanse nego kod Fibonačijevog.

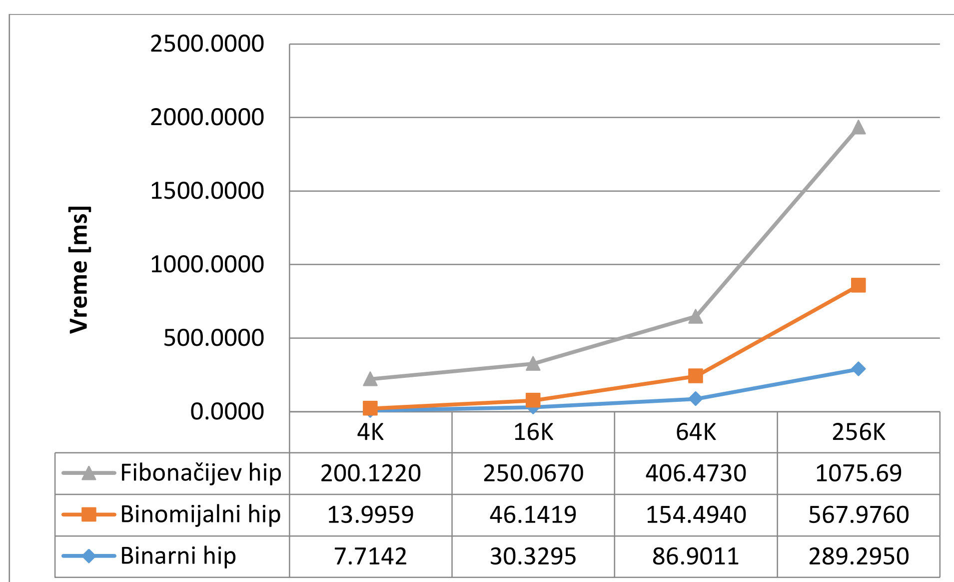
Kod Fibonačijevog hipa, i rada sa ovom operacijom, proces konsolidacije hipa daje najveće usporenje. Iako postoji pokazivač na čvor sa minimalnom vrednošću ključa, a kod binomijalnog hipa se radi pretraga po *root* listi, pokazalo se da na ovu operaciju taj detalj nema velikog uticaja.



	64K	256K	1M	4M	16M
Fibonačijev hip	49.1699	65.2284	111.2260	250.0670	778.9830
Binomijalni hip	29.9382	35.5190	44.1135	46.1419	53.4552
Binarni hip	18.2286	20.9314	25.1006	30.3295	33.0638

Slika 5.6 - Zavisnost vremena izvršavanja od ukupnog broja elementa kod operacije brisanja čvora sa najmanjom vrednosti ključa

Slika 5.6, je prikazana za slučaj sa 16k brisanja čvorova sa najmanjim ključem, a kako bi se video uticaj vremena na ukupan broj brisanja, analiziran je i slučaj sa različitim brojem brisanja, za hip sa 4M čvorova. I u tom slučaju rezultati su dosta slični, odnosno binarni hip ima najbolje performanse, a Fibonačijev najlošije, što se može videti na slici 5.7. Takođe, može se primetiti da se očekivano razlika između binarnog i binomijalnog hipa, sa povećanjem broja brisanja, dodatno i povećava.



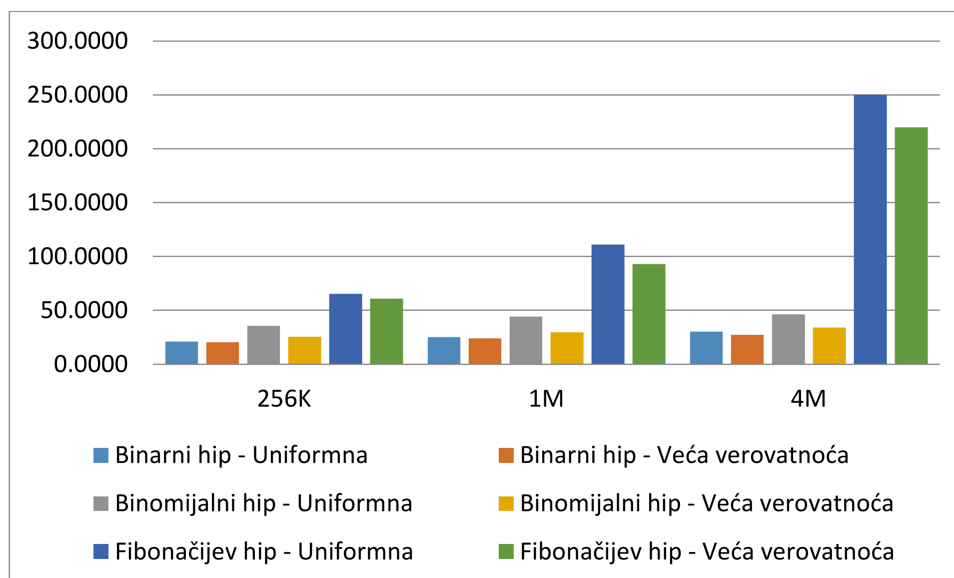
	4K	16K	64K	256K
Fibonačijev hip	200.1220	250.0670	406.4730	1075.69
Binomijalni hip	13.9959	46.1419	154.4940	567.9760
Binarni hip	7.7142	30.3295	86.9011	289.2950

Slika 5.7 - Zavisnost vremena izvršavanja od ukupnog broja izvršenja operacije kod operacije brisanja elementa sa najmanjom vrednosti ključa

Kod ove operacije, uticaj ulazne raspodele se može primetiti kod binomijalnog i Fibonačijevog hipa, što se vidi na slici 5.8. Po definiciji, odnosno već prikazanom načinu umetanja novih elemenata kod binomijalnog hipa i izabranoj ulaznoj raspodeli gde elementi koji su skoro javili imaju veću verovatnoću novog javljanja, javlja se situacija da veći broj čvorova ima ključeve

koji su minimalni, tako da se svi ti ključevi nalaze u okviru *root* liste. Kao što se u analazi implementacije, odnosno opisu ove operacije u binomijalnom hipu može videti, jedan od koraka je i pronalaženje pokazivača na čvor sa minimalnim ključem, odnosno njegovog prethodnog čvor. S obzirom da postoji veći broj čvorova sa istim ključem, i da u tom slučaju nema potrebe ulaziti u setovanje čvora sa minimalnim ključem odnosno njegovog prethodnika, ispostavilo se da ovo može imati mali uticaj na brzinu izvršavanja.

Sličan uticaj se javlja i kod Fibonačijevog hipa. U okviru metode *consolidate*() koja se poziva prilikom izvršenja ove operacije, pre izvršenja metode *fibHeapLink* dolazi do zamena dva čvora u zavisnosti od vrednosti ključa, tako da se uvek kao prvi argument u metodu *fibHeapLink* prosleđuje čvor sa većom vrednošću ključa, a kao drugi čvor sa manjom vrednošću. Kod ulazne raspodele sa većom verovatnoćom javljanja prethodnih ključeva, ređe dolazi do ove zamene pre izvršenja metode *fibHeapLink*. Nakon ove metode, čvor prosleđen kao drugi argument će biti roditelj čvoru prosleđenom kao prvi argument.

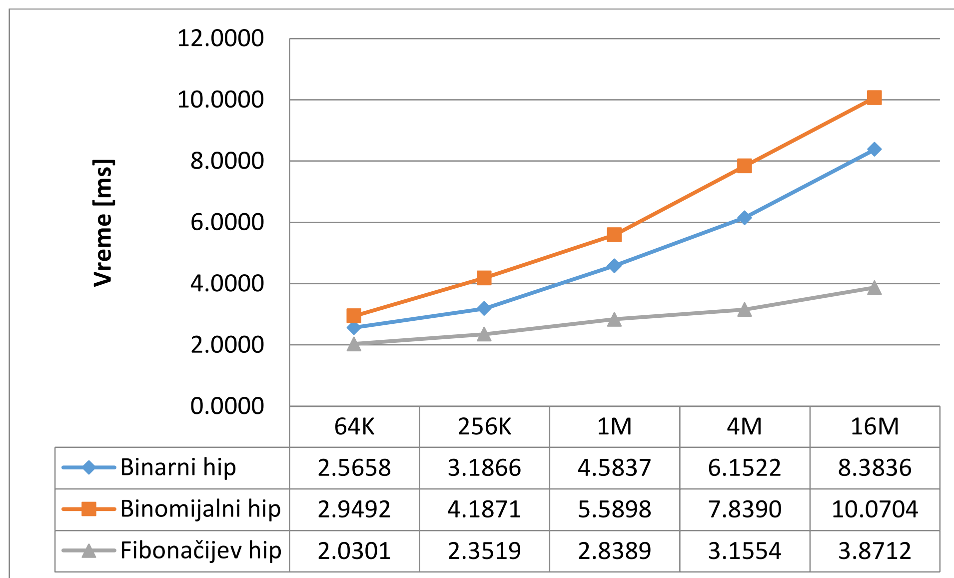


Slika 5.8 - Zavisnost vremena izvršavanja od ukupnog broja elementa pri poređenju različitih ulaznih raspodela kod operacije brisanja elementa sa najmanjim ključem

5.1.4. Smanjivanje vrednost ključa

Smanjivanje vrednosti ključa kod binarnog i binomijalnog hipa ima vremensku složenost *O(log n),* dok je kod Fibonačijevog hipa ta složenost konstantna *O(1).* Da Fibonačijev hip ima najbolje vremenske performanse, pokazalo se i prilikom merenja vremena izvršavanja, koje prikazuje slika 5.9. Ono što je zanimljivo je da iako je složenost konstanta, sa povećanjem veličine hipa, postepeno se povećava i vreme izvršavanja ove operacije. Razlog za takvu situaciju je setovanje flega *mark*, odnosno, kao što je objašnjeno u opisu ove operacija u poglavlju 2.3, ako je roditelj nekog čvora markiran, roditelj se takođe prebacuje u *root* listu. To propagiranje može obuhvatiti veći broj čvorova. Kod binarnog i binomijalnog hipa, ova operacija se izvršava na dosta

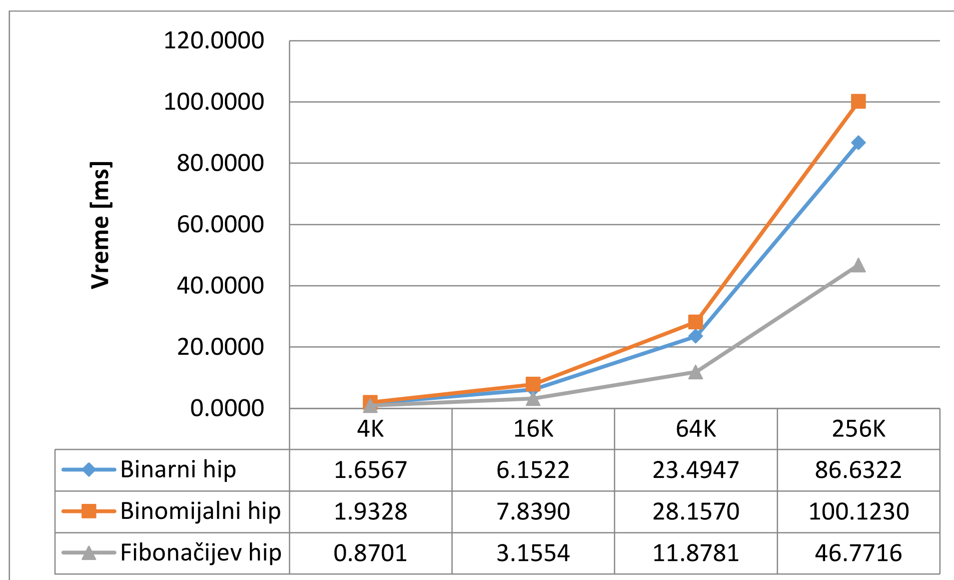
sličan način. Nakom smanjivanja vrednosti ključa, element se premešta ka vrhu hipa. Ipak, sama operacija *shiftUp* kod binarnog hipa je malo brža od zamene pokazivača na relacija roditelj – dete kod binomijalnog hipa. Merenje na slici 6.6. je rađeno sa 16K smanjivanja vrednosti ključa.



	64K	256K	1M	4M	16M
Binarni hip	2.5658	3.1866	4.5837	6.1522	8.3836
Binomijalni hip	2.9492	4.1871	5.5898	7.8390	10.0704
Fibonačijev hip	2.0301	2.3519	2.8389	3.1554	3.8712

Slika 5.9 - Zavisnost vremena izvršavanja od ukupnog broja elementa kod operacije umanjenja vrednosti ključa

Slika 5.10 prikazuje zavisnost vremena izvršavanja od ukupnog broja izvršavanja ove operacije na hipu od 4M elemenata. Rezultati su očekivano slični kao u prethodnoj tabeli.

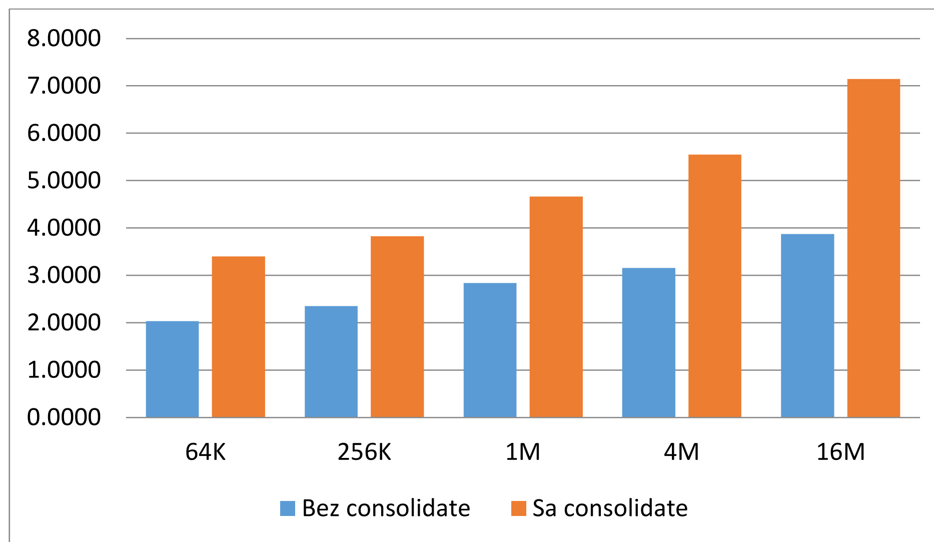


	4K	16K	64K	256K
Binarni hip	1.6567	6.1522	23.4947	86.6322
Binomijalni hip	1.9328	7.8390	28.1570	100.1230
Fibonačijev hip	0.8701	3.1554	11.8781	46.7716

Slika 5.10 - Zavisnost vremena izvršavanja od ukupnog broja umanjenja vrednosti ključa kod operacije umanjenja vrednosti ključa

Izvršenje ove operacije kod Fibonačijevog hipa ima i dodatnu karakteristiku. Ako pre poziva ove operacije, nije rađena nijednom operacija uzimanja najmanjeg elementa, odnosno konkretno, relativno skoro nije pozivana metoda *consolidate()*, većina, odnosno svi čvorovi će biti u okviru *root* liste, što automatski znači da se ne ulazi u deo gde se pozivaju metode *cut* i *cascadingCut*. U okviru prethodna dva merenja u okviru analize ove operacije, rađena su merenja

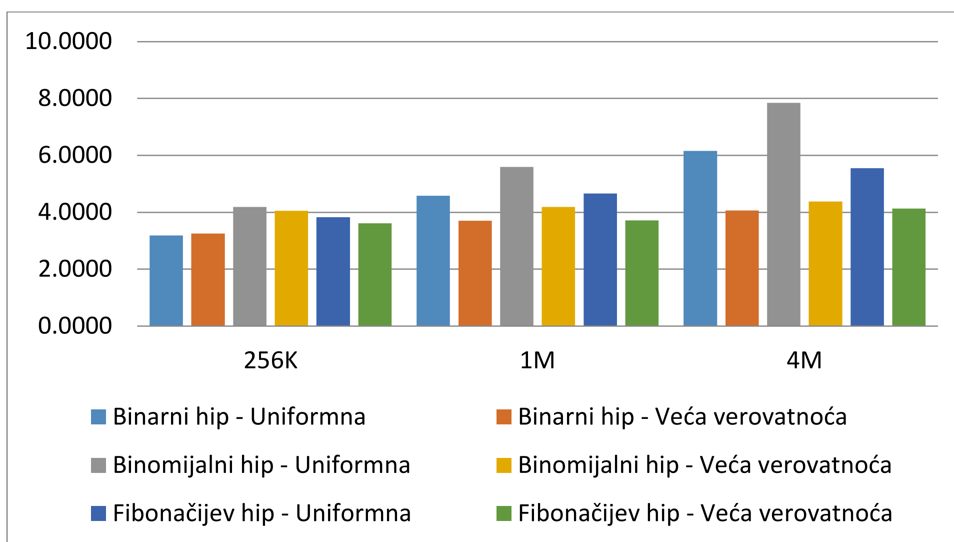
direktno nakon umetanja novih čvorova, a slika 5.11 pokazuje razliku ove situacije, i situacije kada je pre umanjena vrednosti ključa pozvana operacija uzimanja najmanjeg ključa, odnosno, izvršena je *consolidate()* metoda.



Slika 5.11 - Zavisnost vremena izvršavanja od ukupnog broja elemenata kod Fibonačijevog hipa i operacije umanjenja vrednosti ključa, u zavisnosti da li je neposredno pre izvršenja operacije pozvana operacija uzimanja najmanje ključa

Slika 5.12 odnosi se na analizu vremenske zavisnosti u odnosu na način generisanja ključeva. Za razliku od prethodnih operacija, u okviru ove, razlika je vidljiva kod svih vrsta hipova. Osnovni razlog za razlike je način generisanja kod ulazne raspodele gde elementi koji su se skoro javili imaju veću verovatnoću javljanja u odnosu na ostale elementi. Konkretno, kod binarnog hipa, pri metodi *shiftUp*, koja se poziva pri ovoj operaciji, duplo češće se kod uniformnog načina generisanja javlja situacija da ključ elementa roditelja ima veći ključ od elementa deteta. Slično je i kod binomijalnog hipa, gde se isto mnogo češće ulazi u zamenu čvora roditelja sa čvorom deteta kod uniformne raspodele.

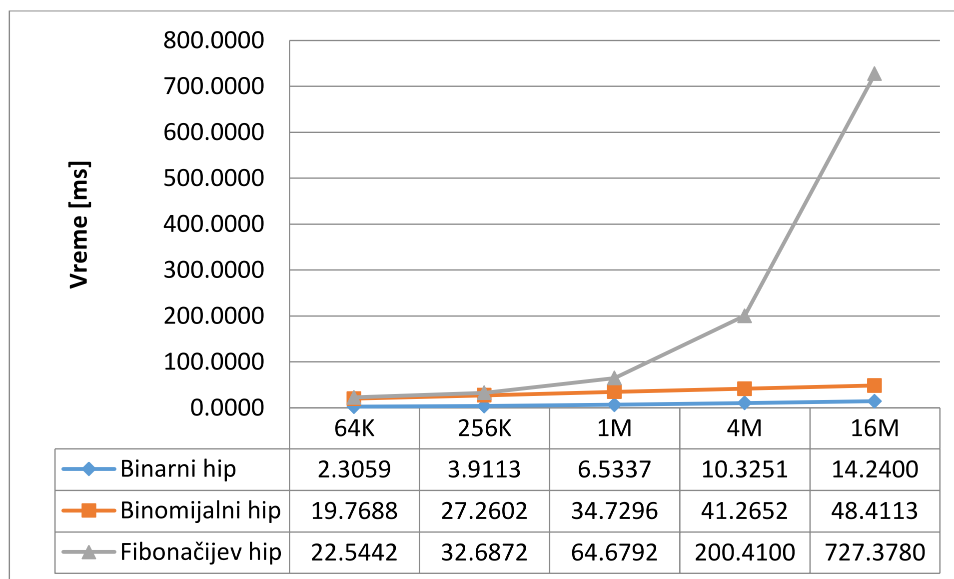
Kod Fibonačijevog hipa, pri situaciji da nije skoro rađena operacija uzimanja čvora sa najmanjim ključem, uticaj ulazne raspodele nije vidljiv. Međutim, ako je skoro pozivana ta operacija, kao što se vidi na narednoj slici, javlja se razlika. Ova razlika se ogleda u mnogo češćem pozivanju metode *cut* kod uniformne ulazne raspodele.



Slika 5.12 - Zavisnost vremena izvršavanja od ukupnog broja elementa pri poređenju različitih ulaznih raspodela kod operacije umanjenja vrednosti ključa

5.1.5. Brisanje elementa

Brisanje elementa kod sve tri vrste hipa ima vremensku složenost *O(log n)*. Ipak, nakon merenja vremena zavisnosti od ukupnog broja elemenata u hipu, pokazalo se da najbolje vreme daje binarni, odnosno binomijalni hip, a najlošije vreme je kod Fibonačijevog hipa, što se vidi na slici 5.13. Na osnovu ovoga, može se zaključiti da su dosta brze metode *shiftUp* odnsno *shiftDown* kod binarnog hipa, odnosno, kod Fibonačijevog hipa veliku ulogu ima metoda *consolidate()* iz operacije uzimanja minimalnog elementa, koja popriličnо usporava i samu operaciju brisanja.

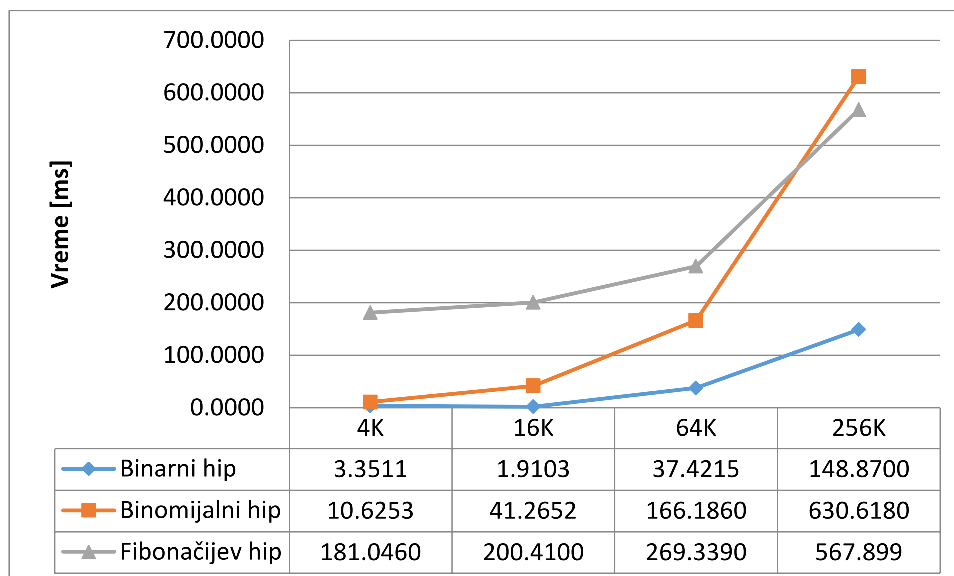


	64K	256K	1M	4M	16M
Binarni hip	2.3059	3.9113	6.5337	10.3251	14.2400
Binomijalni hip	19.7688	27.2602	34.7296	41.2652	48.4113
Fibonačijev hip	22.5442	32.6872	64.6792	200.4100	727.3780

Slika 5.13 - Zavisnost vremena izvršavanja od ukupnog broja elementa kod operacije brisanja elementa

Sa povećanjem broja brisanja, na veličini hipa od 4M elemenata, rezultati su malo drugačiji od očekivanih koji su uzeti sa slike 5.13. Povećanjem broja brisanja, binarni hip očekivano daje najbolja vremena, međutim, kako se povećava broj brisanja, razlika između binomijalnog i Fibonačijevog hipa je sve manja, da bi kod 256K brisanja, Fibonačijev hip imao bolje rezultate od

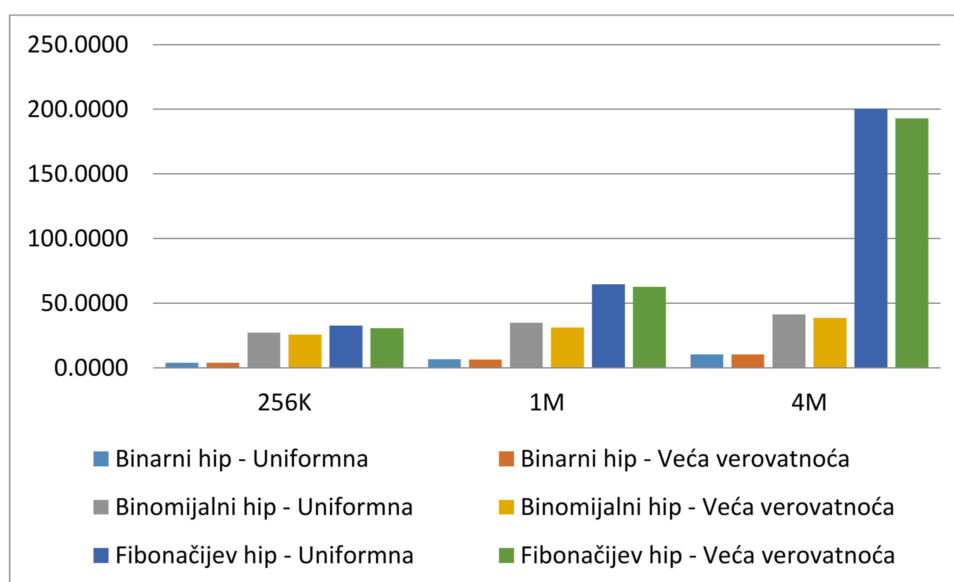
binomijalnog, što prikazuje slika 5.14. Razlog za takve rezultate se može tražiti i u drugoj operaciji koja se koristi kod brisanja elemenata, a to je operacija umanjenja vrednosti ključa, koja ima mnogo bolje performanse kod Fibonačijevog u odnosu na binomijalni hip.



	4K	16K	64K	256K
Binarni hip	3.3511	1.9103	37.4215	148.8700
Binomijalni hip	10.6253	41.2652	166.1860	630.6180
Fibonačijev hip	181.0460	200.4100	269.3390	567.899

Slika 5.14 - Zavisnost vremena izvršavanja od ukupnog broja brisanja elemenata

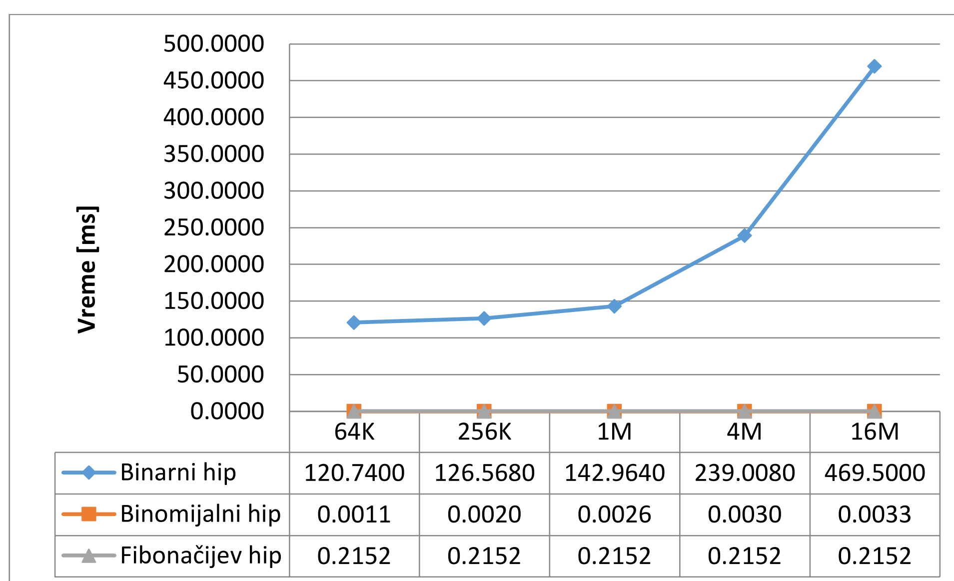
Slika 5.15 prikazuje uticaj izbora ulazne raspodele prilikom izvršenja ove operacije. Ovaj uticaj kod binarnog hipa nije uočljiv, dok kod binomijalnog i Fibonačijevog, s obzirom da se pozivaju operacije smanjenja vrednost ključa, odnosno uzimanja ključa sa najmanjom vrednosti, može primetiti da ova operacija ima bolje vremenske performanse kod ulazne raspodele gde verovatnoća javljanja novih ključeva zavisi od izbora poslednjih nekoliko.



Slika 5.15 - Zavisnost vremena izvršavanja od ukupnog broja elementa pri poređenju različitih ulaznih raspodela kod operacije brisanja elementa

5.1.6. Unija hipova

Merenje rezultata kod ove operacije se radilo spajanjem hipa veličine 4M, dok je veličina drugog hipa varirala, po veličinama koje su se koristile i kod prethodnih operacija. Najlošiju vremensku složenost, operacija unije dva hipa ima kod binarnog hipa, što se prilikom merenja pokazalo kao tačno što i prikazuje slika 5.16. Kod binomijalnog hipa, koji ima logaritamsku vremensku složenost, vreme izvršavanje se pokazalo dosta dobro, a razlog toga leži u samom načinu unije dva hipa, gde je osnova tog algoritma spajanje *root* listi, koje imaju veoma mali broj elemenata. Fibonačijev hip, koji ima konstantno vreme izvršavanja, dao je očekivane rezultate.



	64K	256K	1M	4M	16M
Binarni hip	120.7400	126.5680	142.9640	239.0080	469.5000
Binomijalni hip	0.0011	0.0020	0.0026	0.0030	0.0033
Fibonačijev hip	0.2152	0.2152	0.2152	0.2152	0.2152

Slika 5.16 - Zavisnost vremena izvršavanja od ukupnog broja elemenata kod unije dva hipa

Izbor ulazne raspodele kod ove operacije nema uticaj na vreme izvršavanja operacije.

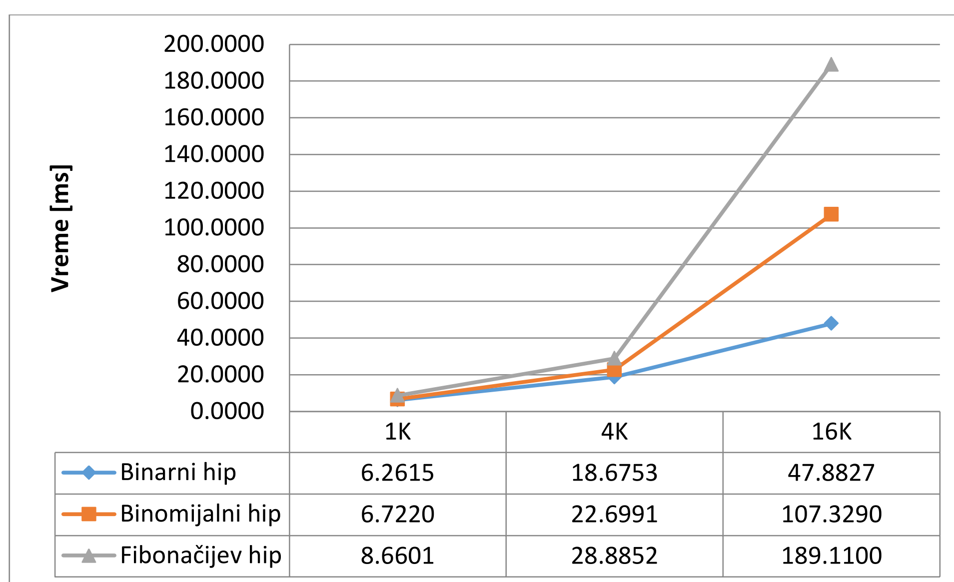
5.2. Simulacije algoritama

U okviru ovog dela, izvršeno je simuliranje dva algoritma koja koriste hipove kao pomoćnu strukturu. U pitanju su Dijkstra algoritam za pronalaženje najkraćeg puta u grafu, kao i algoritam za Hafmanove kodove.

5.2.1. Dijkstra algoritam

Pri simulaciji Dijkstra algoritma, korišćene su tri operacije: umetanje elemenata tako da prvi element ima ključ 0, a ostali maksimalnu vrednost, brisanje najmanjeg ključa sve dok se ne iskoriste svi čvorovi iz grafa, i nakon brisanja najmanjeg ključa nekoliko izvršavanja operacije umanjenja vrednosti ključa. Slika 5.17 prikazuje zavisnost vremena izvršavanja od ukupnog broja elemenata, a kao najbolje performanse, dao je binarni hip, dok je najlošije imao Fibonačijev.

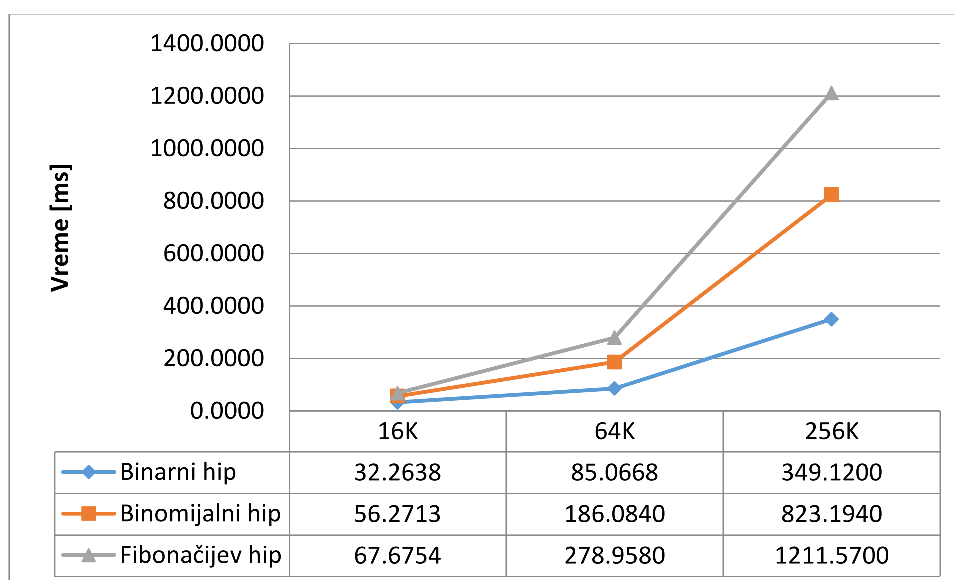
Razlog za takve rezultate se može tražiti u sporosti operacije brisanja ključa sa najmanjom vrednosti, odnosno konkretno metodi *consolidate*().



Slika 5.17 - Zavisnost vremena izvršavanja od ukupnog broja elemenata pri simulaciji Dijkstra algoritma

5.2.2. Hafmanovi kodovi

Za simulaciju ovog algoritma korišćene su dve operacije: dvostruko uzimanje, odnosno brisanje elementa sa najmanjim ključem i nakon toga umetanje novog elementa sa vrednosti ključa koja se dobija sabiranjem dva uzeta ključa. Analiza rezulta je dosta slična kao pri simulaciji Dijkstra algoritma, s tim što je u okviru ovog algoritma još više izraženija prednost binarnog hipa, što i prikazuje slika 5.18.



Slika 5.18 - Zavisnost vremena izvršavanja od ukupnog broja elemenata pri simulaciji Hafmanovog algoritma

6. ZAKLJUČAK

Prioritetni redovi su veoma bitna struktura podataka koja se koristi pri implementaciji mnogih značajnih algoritama. Sama implementacija prioritetnih redova je moguća na više različitih načina, a jedna od najefikasnijih implementacija koristi hip. Postoji više različitih varijanti ove konceptualne strukture podataka. Cilj ovog rada je bio teorijska i implementaciona analiza najrelevantijih vrsta hipova, kao i njihova uporedna evaluacija.

Na početku rada, dat je detaljan opis hip strukture podataka, klasifikacija i analiza tri izabrane vrste hipa: binarnog, binomijalnog i Fibonačijevog hipa. Svaki od izabranih hipova sadrži opis i karakteristike, posle čega je usledila analiza osnovnih operacija koje se najčešće koriste pri radu sa hipovima. Nakon analize izabranih hipova, dat je prikaz najpoznatijih algoritama koji koriste hip strukturu podataka.

Opisu implementacije hipova i osnovnih operacija posvećeno je poglavlje nakon analize hipova, nakon čega je usledila detaljna analiza rezultata dobijenih nakon izvršavanja izabranih operacija. Analiza rezultata je rađena iz više različitih aspekata, kao što su zavisnost vremena izvršavanja od ukupnog broja elemenata u hipu sa fiksnim brojem izvršenja operacija, zavisnost vremena izvšavanja od broja izvršavanja operacija sa fiksnim ukupnim brojem elemenata u hipu kao i analiza u zavisnosti od ulazne raspodele ključeva. Neke od operacija, kao što su umetanje novog elementa kod binomijalnog ili smanjivanje vrednosti ključa kod Fibonačijevog hipa su imali i dodatne analize.

Analiziranjem rezultata izvršavanja operacija kod sve tri vrste hipa, može se reći da je njihova implementacija bila uspešna. Svaki od dobijenih rezultata je imao teorijsku potporu pri

procesu obrade. S obzirom da je osnovna primena hipova u okviru algoritama, kao podaci koji su dosta releventni za analizu implementacije, mogu se uzeti podaci iz dela gde se radila simulacija Dijkstra i Hafmanovog algoritma. Po tim rezultatima, kao hip sa najboljim performansama se pokazao binarni hip. Međutim, s obzirom da su algoritmi koji su uzeti za simuliranje radili sa samo tri različite operacije, a uzevši u obzir da kod operacije kao što je unija hipova binarni hip daje mnogo lošije performanse od ostale dve izabrane vrste, rezultati bi u okviru nekih drugih algoritama mogli biti znatno drugačiji.

Fibonačijev hip je očekivano, kod operacija gde ima konstantu vremensku složenost algoritama, kao što su umetanje elementa, nalaženje elementa sa najmanjim ključem, umanjenje vrednosti ključa ili unija hipova, dao najbolje perfomanse. Međutim, velike prednosti iz tih operacija se gube sa dosta lošim perfomansama kod operacije brisanja elementa sa najmanjim ključem. Binomijalni hip je kroz skoro sve operacije, sem umetanja novog elementa, pokazao dosta dobre performanse.

Dalji razvoj i analiza hipova mogao bi se skoncentrisati na poboljšanje performansi kod nekih operacija za različite vrste hipova, kao što je poboljšanje prilikom operacije umetanja novog elementa kod binomijalnog hipa ili metode *consolidate* prilikom brisanja najmanjeg elementa kod Fibonačijevog hipa. Pored toga, u proces analize se mogu uključiti i dodatne vrste hipova, sa svojim karakteristikama koje u nekim operacijama mogu dati bolje rezultate od, u ovom radu, izabrane tri vrste. Analiza bi se mogla raditi sa još većim brojem elemenata, posebno pri simulaciji algoritama, a u proces analize bi se mogli uključiti i dodatni algoritmi koji koriste hip strukturu podataka.

7. LITERATURA

Svi izvori koje je autor koristio u toku izrade ovog rada su ovde navedeni redom kojim su referencirani u radu. Za svaki navedeni izvor je priložena Internet adresa, ako je bila dostupna u trenutku pravljenja ovog spiska. Izvori kojima autor nije imao direktan pristup, navedeni su neposredno iza reference prema direktno dostupnom izvoru iz koga je informacija preneta, unutar istog rednog broja. Kako bi bilo izbegnuto nepotrebno uvećavanje broja izvora, zajednički redni broj je upotrebljen i za grupisanje izvora koji se odnose na suštinski istu stvar (specifikaciju, program i slično). Format je usklađen sa IEEE uputstvom za navođenje referenci, datim u [22][22][22].

[1] Sedgewick, R., Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Addison-Wesley Professional, 3rd edition, 1998.

[2] Cormen, T., Leiserson, C., Rivest, R., Stein, C., Introduction to Algorithms, The MIT Press, 3rd edition, 2009.

[3] Cormen, T., Leiserson, C., Rivest, R., Stein, C., Introduction to Algorithms, The MIT Press, 2nd edition, 2001.

[4] Tomašević, M., Algoritmi i strukture podatka, Akademska misao Elektrotehnički fakultet, Beograd, 2008.

[5] Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E., The Pairing Heap: A New Form of Self-Adjusting Heap, Alghorithmica, 1986.

[6] Takaoka, T., Theory od 2-3 Heaps, Department of Computer Science, University of Canterbury, New Zealend, 1999.

[7] Graham, R. L, Hell, P., Annals of the History of Computing, IEEE, 1985.

[8] Tarjan, R., Data Structures and Network Algorithms, Society for Industrial and Applied Mathematics, 1987.

[9] Mehlhorn, K., Sanders, P., Algorithms and Data Structures: The Basic Toolbox, Springer, 2008 edition, 2008.

[10] Knuth, D., The Art of Computer Programming, Volume 3, Searching and Sorting, Addison-Wesley, 1998.

[11] Skiena, S., The Algorithm Design Manual, Second Edition, Springer-Verlag, London, 2008.

[12] Kozen, D.C., The Design and Analysis of Algorithms (Monographs in Computer Science), Springer, 1992.

[13] Roche D.L., Experimental Study of High Performance Priority Queues, 2007.,

ftp://ftp.cs.utexas.edu/pub/techreports/honor_theses/cs-07-34-roche.pdf

[14] Fredman, M., Tarjan, R., Fibonacci heaps and their uses in improved network optimization algorithms, ACM New York, 1987.

[15] Goldberg, A., Tarjan, R., Expected Performance of Dijkstra's Shortest Path Algorithm, NEC Research Institute, 1996.

[16] Microsoft Visual Studio, Wikipedia, 2013.,

http://en.wikipedia.org/wiki/Microsoft_Visual_Studio

[17] Intel® Core™ i3-2100 Processor, Intel Corporation, 2013.,

http://ark.intel.com/products/53422

[18] COS 423, Theory of Algorithms, Spring 2013, Princeton University, 2016., https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/BinomialHeaps-2x2.pdf

[19] CSE 548-01 (#84542), AMS 542-01 (#84639): Analysis of Algorithms, Spring 2015, Stony Brook Universitz, 2016., http://www3.cs.stonybrook.edu/~rezaul/Spring-2015/CSE548/CSE548-lectures-16-17-18.pdf

[20] COS 423, Theory of Algorithms, Spring 2013, Princeton University, 2016., https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/FibonacciHeaps.pdf

[21] Huffman coding example, Wikipedia, 2016.,

https://upload.wikimedia.org/wikipedia/commons/d/d0/Huffman_coding-example.svg

[22] IEEE, Preparation of Papers for IEEE Transactions and Journals, http://www.ieee.org/portal/cms_docs_iportals/iportals/publications/journmag/transactions/TRANS-JOUR.doc, 2007.