Sveučilište u Zagrebu
Prirodoslovno - matematički fakultet
Matematički odsjek

Oblikovanje i analiza algoritama - projektni zadatak

Fibonaccijeva hrpa

Roberto Grabovac

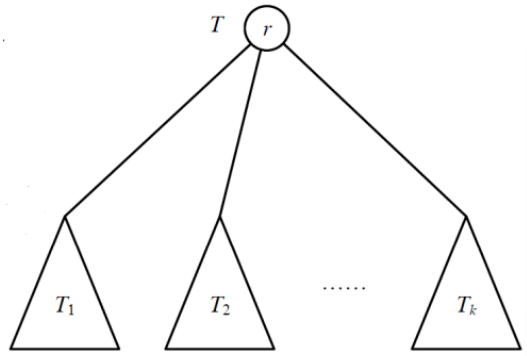
Zagreb, prosinac 2022.

Sadržaj

1 Uvod

Prioritetni red je jedna od temeljnih struktura podataka u računarstvu koja održava hijerarhiju među svojim elementima određenu njihovim prioritetima. Postoje razne implementacije koje su nastale u svrhu poboljšavanja asimptotskih ocjena složenosti operacija koje se na toj strukturi podataka koriste. U praksi se najčešćom i najefikasnijom implementacijom pokazala ona koja koristi *svojstvo hrpe*. Kako bismo precizno definirali prethodno svojstvo, potrebno je uvesti nekoliko pomoćnih definicija.

Definicija 1. ***Stablo*** *T je neprazni konačni skup podataka istog tipa koje zovemo* ***čvorovi****. Pritom postoji jedan istaknuti čvor r koji se zove* ***korijen*** *od T, dok ostali čvorovi grade konačni niz $(T_1, T_2, ..., T_k)$ od 0 ili više disjunktnih (manjih) stabala.*



Slika 1: Stablo T
(Slika je preuzeta iz [11])

Napomena 1. *Svakom čvoru pridružujemo oznaku koja predstavlja određenu informaciju. U skladu s tim, mogu postojati dva čvora s istom oznakom. Smatramo da oznake pripadaju skupu na kojem je definiran totalni uređaj. Nadalje, čvor može imati roditelja, djecu i braću koji su također čvorovi. Čvor bez roditelja nazivamo korijen.*

Sada imamo sve potrebno za preciznu definiciju svojstva hrpe.

Definicija 2. *Podatkovna struktura čija je reprezentacija temeljena na stablu zadovoljava* ***svostvo hrpe*** *ako je oznaka čvora stabla manja ili jednaka od oznaka njegove djece. Takvu strukturu nazivamo* ***hrpa****.*

U ovom radu će biti izložena spomenuta efikasna implementacija koja koristi svojstvo hrpe. Prioritetni red će tada po definiciji biti hrpa pa će njegovi elementi biti čvorovi čije će oznake zapravo predstavljati njihove prioritete. Kao i svaka struktura podataka, prioritetni red podržava nekoliko operacija koje ujedno oslikavaju njegove mogućnosti, odnosno samu funkcionalnost:

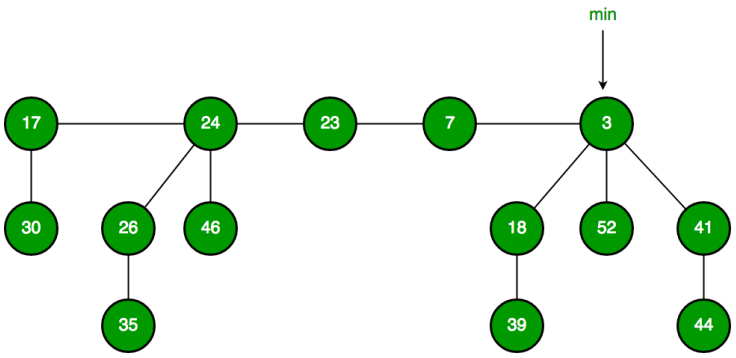
- *insert(x)* : dodaje element x
- *min()* : vraća najmanji element
- *deleteMin()* : briše najmanji element
- *decreaseKey(x, k)* : smanjuje vrijednost prioriteta (oznake) elementa x na k uz pretpostavku da je k manji od trenutnog prioriteta.

Cilj je efikasno implementirati prethodno navedene operacije, a to ćemo učiniti tako da prioritetni red predstavimo **Fibonaccijevom hrpom**.

2 Fibonaccijeva hrpa

Definicija 3. ***Fibonaccijeva hrpa*** *(ili* ***F-hrpa****) je kolekcija međusobno disjunktnih*[1] *stabala koja zadovoljavaju svojstvo hrpe.*

Stabla koja čine Fibonaccijevu hrpu mogu poprimati proizvoljan oblik, a moguća je i situacija da je svako stablo zapravo jedan čvor. Samu Fibonaccijevu hrpu poistovjećujemo s pokazivačem na element (čvor) koji sadrži najmanju oznaku. Jasno, sva stabla moraju biti povezana u cjelinu, kao i



Slika 2: Primjer jedne Fibonaccijeve hrpe
(Slika je preuzeta iz [1])

[1]Svaki element Fibonaccijeve hrpe nalazi se u točno jednom stablu.

elementi pojedinog stabla. Način kako će se ta povezanost ostvariti, a da se potrebne operacije izvršavaju efikasno jest korištenje dvostruke cirkularne vezane liste.

3 Implementacija

Implementacija Fibonaccijeve hrpe bit će napravljena u programskom jeziku C. Čvor ćemo definirati kao strukturu, a Fibonaccijevu hrpu poistovjetiti s pokazivačem na čvor s najmanjom oznakom. Ipak, prije toga je potrebno promotriti karakteristike dvostruke cirkularne vezane liste.

3.1 Dvostruka cirkularna vezana lista

Jednostavna vezana lista (engl. *Singly Linked List*) je linearna struktura podataka. Svaki čvor sadrži pokazivač na svog sljedbenika.[2] Nedostatak jednostavne vezane liste je "nespretno" brisanje elementa jer su potrebna dva pokazivača na čvorove liste kako bi se obavila ta operacija. Također, pretraživanje čvorova omogućeno je samo u jednom smjeru. Alternativa koja rješava prethodno navedene probleme jest dvostruka vezana lista. Glavna strukturna razlika u odnosu na jednostavnu vezanu listu jest ta što svaki čvor, uz pokazivač na svog sljedbenika, sadrži i pokazivač na svog prethodnika.[3] Dodatna prednost je i konstantno vrijeme izvršavanja operacije brisanja elementa ako je poznata njegova adresa. No, dvostruka cirkularna vezana lista predstavlja njezino dodatno poboljšanje. Strukturna razlika je samo u tome što pokazivač na prethodnika prvog čvora sadrži adresu zadnjeg čvora. Slično, pokazivač na sljedbenika zadnjeg čvora sadrži adresu prvog čvora. Takva struktura vezane liste omogućava sekvencijalno pretraživanje cijele liste s početkom u bilo kojem čvoru. Također, moguće je u konstantnom vremenu izvršavanja prijeći s početnog na krajnji čvor, i obratno.

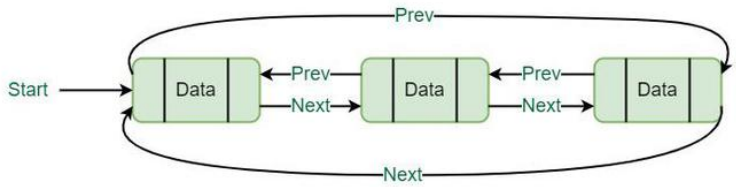
3.2 Struktura Fibonaccijeve hrpe

Sada ćemo opisati kako ostvarujemo povezanost korijena disjunktnih stabala dvostrukom cirkularnom vezanom listom. Struktura *node* (prikaz na slici 4) predstavlja element, odnosno čvor F-hrpe koji ima sljedeće varijable:

- $x.parent$: pokazivač na roditelja čvora x. Ako roditelj ne postoji, postavljen je na NULL

[2] Pokazivač zadnjeg čvora je postavljen na NULL.

[3] Pokazivač na prethodnika početnog čvora je postavljen na NULL, kao i pokazivač na sljedbenika zadnjeg čvora.



Slika 3: Struktura dvostruke cirkularne vezane liste
(Slika je preuzeta iz [4])

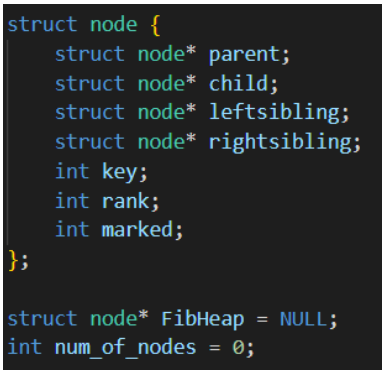
- $x.child$: pokazivač na dijete čvora x. Ako dijete ne postoji, postavljen je na NULL
- $x.leftsibling$: pokazivač na lijevog brata čvora x. Ako lijevi brat ne postoji, onda pokazivač sadrži adresu čvora x
- $x.rightsibling$: pokazivač na desnog brata čvora x. Ako desni brat ne postoji, onda pokazivač sadrži adresu čvora x
- $x.key$: oznaka, prioritet čvora x
- $x.rank$: broj djece čvora x
- $x.marked$: određeno svojstvo čvora x. Poprima vrijednost *false* ili *true*

Pokazivači *leftsibling* i *rightsibling* upravo ostvaruju spomenutu dvostruku cirkluranu vezanu listu. Nadalje, pokazivači *parent* i *child* potrebni su kako bismo povezali i elemente pojedinog stabla koje u uniji s ostalim stablima čini F-hrpu. Uočimo, nisu samo korijeni disjunktnih stabala povezani dvostrukom cirkularnom vezanom listom, već i djeca pojedinog čvora. Naravno, to dolazi iz rekurzivne definicije stabla. Potreba za varijablama *rank* i *marked* bit će jasna iz opisa operacija *deleteMin* i *decreaseKey*. Konačno, Fibonaccijevu hrpu definirali smo kao globalnu varijablu *FibHeap* te će ona u bilo kojem trenutku[4] sadržavati adresu korijena s najmanjom vrijednosti varijable *key*. Iz razloga koji će biti poznat prilikom implementacije *deleteMin*, definirali smo i globalnu varijablu *num_of_nodes* koja predstavlja ukupan broj čvorova F-hrpe.

3.3 Operacije

U nastavku ćemo objasniti implementaciju funkcija navedenih u uvodu. Uz njih su implementirane dodatne funkcije:

[4]Osim kada je prazna. U tom slučaju *FibHeap* = NULL.



Slika 4: Struktura *node*, globalne varijable *FibHeap* i *num_of_nodes*

- *merge(H, sizeof(H))* : spaja F-hrpe H i $FibHeap$
- *delete(x)* : briše čvor x iz F-hrpe *FibHeap*
- *printFibHeap(x)* : ispisuje F-hrpu čiji je korijen s najmanjom oznakom upravo x

Funkcija *min()* je trivijalna jer je potrebno vratiti vrijednost oznake varijable *FibHeap*. Nadalje, nove čvorove ćemo ubacivati tako što ih dodajemo u vezanu listu korijena. Zbog toga je i *insert(x)* trivijalna pa nema potrebe za detaljnijom analizom. Od dodatnih funkcija, trivijalna je *merge*. Naime, ona samo pokazivačima povezuje korijene s najmanjom oznakom dviju F-hrpa na način da se očuva svojstvo hrpe. No, funkcije *deleteMin* i *decreaseKey* su znatno složenije te svaki njihov poziv u velikoj mjeri preoblikuje dotadašnju F-hrpu. Važno je napomenuti kako fleksibilnost u broju i njihovom rasporedu u disjunktnim stablima omogućuje korištenje metode tzv. *lijene evaluacije*, odnosno odgađanja obavljanja vremenski složenih operacija. Nakon objašnjenja funkcija *deleteMin* i *decreaseKey*, vidjet ćemo gdje se točno ta metoda očituje pri radu s Fibonaccijevom hrpom.

3.3.1 deleteMin

Opis funkcije je krajnje jednostavan - briše minimalni element F-hrpe. No, nakon toga je potrebno preurediti hrpu i način na koji se to radi upravo čini cjelokupnu implementaciju izrazito efikasnom. Rad funkcije možemo opisati u tri faze:

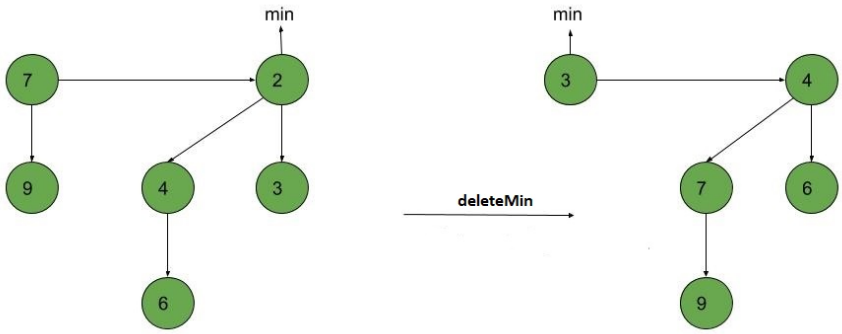
1. Sva djeca čvora s najmanjom oznakom dodavaju se u vezanu listu korijena. Potom se dotični čvor uklanja iz F-hrpe tako što ga jednostavno izbacimo iz vezane liste i oslobodimo memoriju koju je zauzimao.

2. Provodimo *konsolidaciju* korijena - sve dok postoje dva korijena s istim brojem djece (ovdje uočavamo razlog uvođenja varijable *rank* u strukturu *node*), postavi prvi korijen kao dijete drugog (ili obratno) tako da se očuva svojstvo hrpe.

3. Jednim prolaskom kroz novu listu korijena pronađi čvor s najmanjom oznakom i njegovu adresu postavi u varijablu *FibHeap*.

Prvi korak lako rješavamo korištenjem funkcije *merge* onoliko puta koliko je čvor s najmanjom oznakom imao djece. Uočimo da u drugi argument funkcije *merge* (predstavlja broj djece F-hrpe) stavljamo 0, a ne *rank* djeteta jer samo preuređujemo našu F-hrpu, tj. ne dodajemo joj neku sasvim novu hrpu pa da se treba ažurirati ukupan broj čvorova. Drugi korak je dosta složeniji. Kako bi funkcija *deleteMin* bila preglednija, koristimo pomoćnu funkciju *consolidate()* koja provodi opisanu konsolidaciju korijena. Uočimo kako ćemo u toj funkciji učiniti neki korijen djetetom drugog korijena. Iz tog razloga uvodimo još jednu pomoćnu funkciju *link_roots* koja korijen u prvom argumentu postavlja u vezanu listu djece korijena dobivenog u drugom argumentu funkcije. Ta se funkcija može jednostavno implementirati jer sve što se događa jest izbacivanje čvora iz jedne liste i njegovo ubacivanje u drugu listu. Zanimljivo pitanje je kako bismo najpraktičnije mogli provesti konsolidaciju u funkciji *consolidate()*. Rješenje koje je korišteno u implementaciji jest polje pokazivača $node^{**}array$ na korijene F-hrpe *FibHeap*. Na početku sve njegove elemente inicijaliziramo na NULL. Iteriranjem po vezanoj listi korijena ažuriramo stanje u pripadnom polju - ako je broj djece korijena (*rank*) jednak i, onda u $array[i]$ postavljamo adresu tog korijena. Postoje dva slučaja prije upisivanja: $array[i] == NULL$ ili $array[i] \neq NULL$. U prvom slučaju je sve u redu te nastavljamo iteraciju. U drugom pak slučaju imamo da već postoji korijen u *FibHeap* kojem je broj djece i. Tada funkcijom *link_roots* spajamo ta dva korijena tako da se održi svojstvo hrpe, odnosno šaljemo ih kao argumente funkciji *link_roots* u odgovarajućem redoslijedu. Nakon spajanja, postavljamo $array[i] == NULL$, dok u $array[i+1]$ zapisujemo adresu korijena koji je bio u drugom argumentu funkcije *link_roots* i ponavljamo postupak. Jasno, kada završimo s iteracijom, u tom će polju biti pokazivači na sve korijene hrpe *FibHeap* od kojih svaka dva imaju različit broj djece. Konačno, treći korak smo također implementirali u samu funkciju *consolidate()* jer možemo iskoristiti polje *array* za povezivanje postojećih korijena (nastalih nakon drugog koraka) u dvostruku cirkularnu vezanu listu i pri tome pronaći onaj korijen s najmanjom oznakom.
Ono što je u cijeloj priči zanemareno jest potrebna duljina polja *array* za provođenje drugog koraka. Funkcija za duljinu doći će nakon provođenja amortizacijske analize u nastavku, a ono o čemu ovisi jest ukupan broj

čvorova hrpe FibHeap. Time smo ujedno i opravdali uvođenje globalne varijable *num_of_nodes*.



Slika 5: Primjer hrpe nakon djelovanja operacije *deleteMin*
(Slika je preuzeta iz [2])

3.3.2 decreaseKey

Operacija *decreaseKey* radi upravo to - smanjuje vrijednost oznake nekog čvora. Ova operacija je važna zbog svoje primjene u Dijkstrinom i Primovom algoritmu, ali o tome nešto više kasnije. Pri izvršavanju funkcije *decreaseKey* važno je voditi računa o tome da se očuva svojstvo hrpe, to jest da roditeljski čvor ima manju ili jednaku vrijednost oznake od vrijednosti njegove djece. Iz tog razloga imat ćemo nekoliko slučajeva:

1. Nova vrijednost čvora čiju vrijednost smanjujemo je veća ili jednaka od vrijednosti roditeljskog čvora.

2. Nova vrijednost čvora čiju vrijednost smanjujemo manja je od vrijednosti roditeljskog čvora i *marked* roditeljskog čvora je *false*.

3. Nova vrijednost čvora čiju vrijednost smanjujemo manja je od vrijednosti roditeljskog čvora i *marked* roditeljskog čvora je *true*.

Opišimo kako se operacija izvršava za pojedini slučaj:

1. Budući da u ovom slučaju nije narušeno svojstvo hrpe, izvršavanje *decreaseKey* je vrlo jednostavno: postavimo *key* zadanog čvora na novu vrijednost.

2. U ovom slučaju također postavimo *key* zadanog čvora na novu vrijednost, no kako bismo učuvali svojstvo hrpe prebacujemo čvor u listu

korijena. To radimo na očekivani način, namještajući odgovarajuće pokazivače: u slučaju da je zadani čvor bio *child* svog roditelja, za novi *child* postavljamo npr. *righsibling* zadanog čvora, te smanjujemo *rank* roditeljskog čvora za jedan. Također, postavaljamo *marked* roditeljskog čvora na *true*, „spajamo“ *leftsibling* i *rightsibling* zadanog čvora, i ubacujemo zadani čvor u listu korijena na neko mjesto, na primjer desno od dosadašnjeg *FibHeap-a*. Na kraju provjerimo je li nova vrijednost zadanog čvora manja od dosadašnjeg *FibHeap-a*, kojeg po potrebi postavaljamo na zadani čvor. Konačno, ako je *marked* zadanog čvora bio *true*, postavljamo ga na $false$.

3. Ovaj slučaj započinje isto kao i 2. slučaj, ali nakon toga treba i roditeljski čvor na isti način prebaciti u listu korijena budući da mu je *marked true*. Nakon toga provjeravamo je li *marked* roditelja od roditelja originalnog čvora *true*, i ako jest onda ponavljamo postupak sve dok ne dođemo do „vrha“ ili *marked* nekog roditeljskog čvora u postupku bude *false*, te ga postavljamo na *true*. Naravno, svakom čvoru roditelju kojeg premještamo u listu korijena postavljamo $marked$ na $false$.

Dakle, pomoću *marked* osiguravamo da određeni čvor ne izgubi više od jednog djeteta kako bi komponente Fibonaccijeve hrpe bile stabla koja ne odstupaju previše od binomijalnog stabla, što nam je bitno kako bi operacije Fibonaccijeve hrpe imale dobru složenost.

3.3.3 delete, printFibHeap

Operaciju *delete* provodimo kao jednostavnu kombinaciju operacija *decreaseKey* i *deleteMin*. Novu vrijednost čvora kojeg želimo brisati postavimo na neku manju od najmanje vrijednosti čvora u hrpi, npr. $FibHeap \rightarrow key$–1. Računanje te vrijednosti se događa u konstantnom vremenu jer je *FibHeap* uvijek pokazivač na čvor s minimalnom vrijednošću. Ovime smo osigurali da će operacija *deleteMin* izbrisati upravo traženi čvor.
Rekurzivnom funkcijom *PrintFibHeap* se na pregledan način ispisuje stanje hrpe. Ova operacija nije ključna za razumijevanje Fibonaccijeve hrpe, ali bez ispisivanja stanja hrpe bilo bi teško ispitivati valjanost implementacije ostalih operacija.

4 Amortizacijska analiza

Asimptotska analiza vremenske složenosti ponekad može dati prepesimističnu ocjenu, pogotovo ako je riječ o strukturama podataka nad kojima se obično

izvodi čitav niz operacija. Jasno je da jedna operacija može utjecati na vremensku složenost niza operacija koje joj slijede. Problem nastaje ako struktura podataka sadrži npr. jednu vremenski "skupu" operaciju koja se ne izvodi toliko često, dok su sve ostale "jeftine" i češće se pozivaju. Intuitivno je jasno da tada ne bi bilo ispravno reći da je struktura podataka neefikasno implementirana. Stoga, kako bismo dobili realniji zaključak o kvaliteti implementacije koristimo amortizacijsku analizu. U takvom tipu analize potrebno je naći "najgori" redoslijed operacija (gledano u kontekstu vremenske složenosti) te iz toga dobiti prosječno vrijeme izvođenja neke operacije. Tada operacije koje su vremenski "skupe" mogu nakon dobivanja prosječne ocjene ispasti "jeftine".

Primjer 1. *Trgovina sadrži 500 proizvoda čija je cijena 1 kn te samo jedan proizvod cijene 500 kn. Marko želi kupiti sve proizvode, no ne zna koliko mu je potrebno novaca jer nema informaciju o broju proizvoda i njihovim cijenama. Zbog toga je za savjet upitao dva prijatelja kojima je poznato stanje u trgovini. Prvi prijatelj uvijek voli biti jako siguran te mu govori da je dovoljno* $501 \times 500 = 250500$ *kn za kupnju svih proizvoda. Drugi prijatelj pak smatra da je dovoljno* $500 \times 1 + 1 \times 500 = 1000$ *kn. Očito da je ocjena drugog puno bolja od ocjene prvog prijatelja. Prvi prijatelj je koristio asimptotsku, a drugi amortizacijsku analizu problema.*

Fibonaccijeva hrpa je naravno jedna struktura podataka te ćemo u skladu s prethodno navedenim koristiti amortizacijsku analizu za pronalaženje vremenske složenosti definiranih operacija. Postoji nekoliko tehnika u provođenju amortizacijske analize. Ona koju ćemo mi koristiti naziva se *metoda potencijala*.

4.1 Metoda potencijala

Metoda potencijala podrazumjeva početno stanje strukture podataka D_0 nad kojim će biti izvršeno n operacija. Tada dobivamo niz $D_1, D_2, ..., D_n$ gdje D_i predstavlja strukturu podataka nakon obavljanja i-te operacije po redu. Također, definiramo konstantu c_i kao (vremenski) trošak izvođenja i-te operacije. Kako bismo svakom stanju D_i strukture podataka pridružili *potencijal* uvodimo *funkciju potencijala* Φ. Nadalje, definiramo amortizirani trošak i-te operacije:

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Ukupan amortizirani trošak izvođenja n operacija u nizu je stoga:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})). \tag{1}$$

Tehnikom teleskopiranja iz (1) dobivamo da je gornja suma jednaka:

$$\sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \tag{2}$$

Dodatno, u ovoj se metodi zahtjeva $\Phi(D_i) \geq \Phi(D_0), i = 1, ..., n$ kako bi amortizirani trošak niza od n operacija (2) bio gornja ograda za stvarni trošak. Standardno se to radi na način da se definira $\Phi(D_0) = 0$ i dokaže $\Phi(D_i) \geq 0$. Vrijednost $\Phi(D_i)$ shvaćamo kao *potencijal* strukture podataka nakon izvođenja i-te operacije za kojeg se dakle zahtjeva da je uvijek nenegativan. Potencijal se pak može shvatiti kao dodatan "kredit" (nastao iz izvođenja i-te operacije) koji se može iskoristiti za izvođenje iduće operacije. Stoga je razumno za početno stanje pretpostaviti $\Phi(D_0) = 0$. Vrijedi napomenuti kako ova tehnika amortizacijske analize može biti problematična zbog definiranja funkcije potencijala. Do te se funkcije najčešće dolazi empirijski.

4.2 Karakteristike Fibonaccijeve hrpe

U ovom poglavlju ćemo detaljno analizirati ograničenja na strukturu Fibonaccijeve hrpe koja će biti vrlo važna za njezinu amortizacijsku analizu u nastavku. Usput ćemo uočiti i razlog zašto je ovako implementirana hrpa dobila naziv po nadimku talijanskog matematičara Leonarda da Pisa-e.
Prva karakteristika dana je u obliku leme i odnosi se na donju ogradu ranga (broja djece) proizvoljnog čvora.

Lema 1. *Neka je x proizvoljni čvor Fibonaccijeve hrpe i pretpostavimo da je $x.rank = k$. Neka su $y_1, y_2, ..., y_k$ djeca čvora x poredana po redoslijedu ubacivanja. Tada vrijedi $y_1.rank \geq 0$ i $y_i.rank \geq i - 2, i = 2, 3, ..., k$.*

Dokaz. Jasno da $y_1.rank \geq 0$. Za $i \geq 2$ možemo uočiti da u trenutku kada je y_i postao dijete čvora x, čvorovi $y_1, ..., y_{i-1}$ su već bila djeca čvora x. Iz načina djelovanja funkcije *consolidate()*, zaključujemo da je moralo vrijediti $y_i.rank = i - 1$. Nakon spajanja, postoji mogućnost da čvor y_i izgubi najviše jedno dijete u nekom od poziva funkcije *decreaseKey* (inače bi bio prebačen u vezanu listu korijena) pa doista vrijedi $y_i.rank \geq i - 2$.

Sada slijedi definicija Fibonaccijevih brojeva i lema koja je vezana uz jedno njihovo svojstvo. Dokaz leme ćemo izostaviti jer se trivijalno provodi indukcijom.

Definicija 4. *Neka je $k \geq 0$. Fibonaccijeve brojeve F_k definiramo rekurzivnom relacijom: $F_0 = 0, F_1 = 1, F_k = F_{k-1} + F_{k-2}, k \geq 2$.*

Lema 2. *Za svaki $k \geq 0$ vrijedi:*

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

pri čemu konstanta ϕ predstavlja zlazni rez i iznosi $\phi = \frac{1+\sqrt{5}}{2}$.

Originalni autori ovog načina implementacije hrpe navode kako je ime dobila zbog rezultata sljedećeg korolara.

Korolar 1. *Neka je x proizvoljan čvor Fibonaccijeve hrpe i pretpostavimo x.rank = k. Označimo s d(x) broj potomaka čvora x, uključujući i njega samoga. Tada vrijedi:*

$$d(x) \geq F_{k+2} \geq \phi^k.$$

Dokaz. Neka m_k minimalni mogući broj potomaka čvora ranga k. Očito $m_0 = 1, m_1 = 2$. Raspišemo li (koristeći rezultat leme 1) minimalan broj čvorova stabla čiji je korijen ranga k, odnosno vrijednosti za $m_2, m_3, m_4...$ lako naslućujemo da bi moglo vrijediti $m_k = F_{k+2}$. Dokažimo tu slutnju (jakom) matematičkom indukcijom:

- <u>baza</u>: $m_0 = 1 = F_2$, $m_1 = 2 = F_3$
- <u>pretpostavka</u>: pretpostavimo da vrijedi $m_{r-1} = F_{r+1}$ za sve $r < k$ gdje $k \geq 2$ i $k \in \mathbb{N}$
- <u>korak</u>: koristeći rezultat leme 1 raspisujemo m_k na sljedeći način:

$$\begin{aligned} m_k &= 1 + 1 + m_{2-2} + m_{3-2} + ... + m_{k-2} \\ &= 1 + 1 + \sum_{i=0}^{k-2} m_i \\ &= 1 + 1 + \sum_{i=0}^{k-3} m_i + m_{k-2} \\ &= m_{k-1} + m_{k-2} \\ &= F_{k+1} + F_k \\ &= F_{k+2} \end{aligned}$$

pri čemu smo u predzadnjoj jednakosti koristili pretpostavku indukcije.

Prema principu matematičke indukcije, doista vrijedi $m_k = F_{k+2}$. Koristeći lemu 2 dolazimo do tvrdnje korolara:

$$d(x) \geq m_k = F_{k+2} \geq \phi^k.$$

Prisjetimo li se objašnjenja rada funkcije *deleteMin* i njezine pomoćne funkcije *consolidate*, ostala je nepoznata duljina polja *array*. Odgovor na to pitanje daje sljedeći korolar.

Korolar 2. *Maksimalni rang M(x) proizvoljnog čvora Fibonaccijeve hrpe s n elemenata je odozgo ograničen s* $\lfloor log_\phi n \rfloor$.

Dokaz. Neka je x proizvoljan čvor Fibonaccijeve hrpe s n elemenata. Označimo s $k = x.rank$. Tada vrijedi $n \geq d(x) \geq \phi^k$ što je nakon logaritmiranja ekvivalentno s $k \leq log_\phi n$.

4.3 Analiza Fibonaccijeve hrpe

Označimo s F Fibonaccijevu hrpu nad kojom ćemo provesti amortizacijsku analizu. Definiramo funkciju potencijala:

$$\Phi(F) = t(F) + 2m(F)$$

gdje je $t(F)$ broj stabala, $m(F)$ broj označenih čvorova hrpe F. Uočimo da tako definirana funkcija poprima nenegativne vrijednosti, dok iznosi 0 na početnoj hrpi. Zbog toga ćemo biti u mogućnosti amortiziranim vremenom izvršavanja operacija ograničiti stvarno vrijeme izvršavanja niza takvih operacija. U nastavku ćemo izračunati amortizirani trošak svake od operacija za hrpu F.

insert

Neka je F' hrpa nastala dodavanjem čvora u hrpu F. Iz načina implementacije funkcije *insert* slijedi da se broj stabala poveća za 1, dok broj označenih čvorova ostaje isti. Stvarni trošak c je jasno konstanta, odnosno 1. Sada računamo amortizirani trošak operacije *insert*

$$\begin{aligned} a &= c + \Phi(F') - \Phi(F) \\ &= 1 + (t(F) + 1 + 2m(F)) - (t(F) + 2m(F)) \\ &= 2 \end{aligned}$$

te dobivamo da je amortizirana složenost operacije $O(1)$.

merge

Neka je F hrpa nastala spajanjem hrpa F_1 i F_2. Iz imlementacije funkcije *merge* slijedi $t(F) = t(F_1) + t(F_2)$ i $m(F) = m(F_1) + m(F_2)$, dok stvarni trošak c iznosi 1. Sada računamo amortizirani trošak operacije *merge*

$$\begin{aligned} a &= c + \Phi(F) - (\Phi(F_1) + \Phi(F_2)) \\ &= 1 + t(F_1) + t(F_2) + 2(m(F_1) + m(F_2)) - (t(F_1) + 2m(F_1) + t(F_2) + 2m(F_2)) \\ &= 1 \end{aligned}$$

te dobivamo da je složenost operacije $O(1)$.

deleteMin

U analizi ove operacije ćemo izravno koristiti rezultate dokazane u prethodnom potpoglavlju. Kao i obično, neka je $F^{'}$ hrpa nastala nakon brisanja korijena s minimalnom oznakom hrpe F. Izračunajmo prvo stvarni trošak ove operacije. Prvi korak je prebacivanje djece čvora $FibHeap$ u vezanu listu svih korijena. Dokazali smo da je broj djece proizvoljnog čvora reda veličine $d(n) = O(\log n)$, a kako je riječ o vezanoj listi, dodavanje jednog čvora ima red veličine $O(1)$. Konačno, složenost prvog koraka algoritma je $O(\log n)$ te vrijedi $t(F^{'}) = d(n) + t(F) - 1$. U drugom koraku provodimo konsolidaciju, odnosno spajanje stabala čiji korijeni imaju isti rang. Spajanje neka dva stabla u jedno (operacija *link_roots*) izvedivo je u složenosti $O(1)$. Kako se takva spajanja mogu izvršiti najviše $d(n) + t(F) - 1$ puta, stvarni trošak drugog koraka je $O(d(n) + t(F))$. U trećem koraku iteriramo listom korijena i tražimo onaj s najmanjom vrijednosti oznake. Vrijeme izvršavanja proporcionalno je broju stabala kojih nakon drugog koraka može biti najviše $d(n) + 1$. Dakle, složenost trećeg koraka je $O(d(n))$. Konačno, zaključujemo da je stvarna složenost operacije *deleteMin* upravo $O(d(n) + t(F))$. Što se tiče funkcije potencijala, nakon operacije *deleteMin* imamo najviše $d(n) + 1$ korijena, odnosno stabala, dok se broj označenih čvorova ne mijenja. Sada možemo izračunati amortizacijski trošak operacije

$$\begin{aligned} a &= c + \Phi(F^{'}) - \Phi(F) \\ &= O(d(n) + t(F)) + (d(n) + 1 + 2m(F)) - (t(F) + 2m(F)) \\ &= O(2d(n)) + O(t(F)) - t(F) + 1 \\ &= O(d(n)) = O(\log n) \end{aligned}$$

čime dobivamo da je amortizacijska složenost operacije $O(\log n)$.

decreaseKey

U prvom i drugom slučaju opisanom u analizi rada funkcije *decreaseKey*, njezina je stvarna složenost $O(1)$. No, zanima nas najgora složenost te se ona ostvaruje u trećem slučaju. Naime, operacija premještanja proizvoljnog čvora u listu korijena je složenosti $O(1)$. Pretpostavimo da smo premjestili k označenih roditeljskih čvorova. U tom je slučaju stvarna složenost funkcije *decreaseKey* $O(k)$. Za potrebe računanja razlike potencijala, označimo s $F^{'}$ hrpu dobivenu nakon izvršavanja *decreaseKey* nad hrpom F. U skladu s gornjom pretpostavkom, nakon djelovanja operacije moguće je stvaranje

najviše k novih stabala, dok se maksimalno moglo maknuti $k-2$ oznaka - u trećem koraku svakako mičemo najviše $k-1$ oznaka, a dodajemo najviše 1. Sada imamo sve potrebne podatke za računanje amortizacijskog troška operacije

$$\begin{aligned} a &= c + \Phi(F^{'}) - \Phi(F) \\ &= O(k) + (t(F) + k + 2(m(F) - k + 2)) - (t(F) + 2m(F)) \\ &= O(k) + 4 - k = 4 \end{aligned}$$

pa dobivamo da je amortizacijska složenost operacije reda veličine $O(1)$.

delete

Implementacija ove operacije sastoji se od jednog poziva funkcija $decreaseKey$ i $deleteMin$ pa u skladu s njihovom amortizacijskom analizom, zaključujemo da je amortizacijska složenost operacije *delete* reda veličine $O(\log n)$.

5 Primjene Fibonaccijeve hrpe

Prioritetni redovi ili hrpe nastale su prvenstveno radi implementacije Dijkstrinog algoritma. Kako bi se poboljšalo vrijeme izvršavanja algoritma, smišljene su razne implementacije prioritetnih redova, a svojom se efikasnošću ističe Fibonaccijeva hrpa. U teoriji, takva implementacija ima jako dobru amortizacijsku složenost, no u praksi se pokazuje da su neki drugi tipovi hrpa ipak brži. Razlog tome je dosta *skrivene* memorije u implementaciji. Naime, svaki čvor sadrži čak četiri pokazivača, dok ostale hrpe trebaju dva ili tri. U nastavku promatramo tri algoritma koji se baziraju na korištenju Fibonaccijeve hrpe.

5.1 Heap sort

Heap sort je jedan od algoritama za sortiranje. Neka polje koje trebamo sortirati sadrži n elemenata. Način na koji provodimo heap sort je krajnje jednostavan - prvo ubacujemo n elemenata operacijom *insert* u Fibonaccijevu hrpu, a potom n puta koristimo operacije *min* (ispis elementa) i $deleteMin$. Iz analize Fibonaccijeve hrpe, odmah dobivamo (amortiziranu) složenost Heap sort-a:

$$O(n + n + n \cdot \log n) = O(n \cdot \log n).$$

Vidimo da je heap sort teoretski jednako brz kao merge i quick sort (u prosječnom slučaju). No, memorijska složenost je $O(n)$, ali postoji poboljšanje

koje ju svodi na $O(1)$ tako da se sortiranje provodi u mjestu. Ipak, heap sort nije uspio nadmašiti quick sort i merge sort. Merge sort je u prednosti kod jako velike količine podataka jer se može paralelno izvoditi na više računala, dok to kod heap sorta nije slučaj jer podaci nisu međusobno neovisni pri sortiranju kao kod merge sort-a. Quick sort je pak u prednosti jer su podaci koje uspoređujemo uvijek relativno *blizu*, tj. bit će pohranjeni u *cache* memoriju procesora pa će njihova usporedba biti brza. S druge strane, u heap sortu uspoređujemo roditelja i dijete koji ne moraju biti *blizu* u memoriji računala što za posljedicu dovodi do toga da možda neće biti zajedno dohvaćeni i spremljeni u *cache* memoriju procesora koja je kapacitetom dosta mala. Zbog toga će njihova usporedba trajati duže od očekivanog.

5.2 Dijkstrin algoritam

Dan je usmjereni graf $G = (V, E)$ pri čemu V predstavlja skup vrhova, a E skup bridova te je svakom bridu pridružena njegova vrijednost (cijena), odnosno nenegativan broj. Dodatno, definiramo $n \coloneqq |V|$ i $m \coloneqq |E|$. Potrebno je za neki istaknuti vrh $s \in V$ pronaći najkraći (najjeftiniji) put do svakog vrha v grafa G, uz pretpostavku da doista postoje putevi od s do svakog vrha grafa G. Ovaj problem upravo rješava Dijkstrin algoritam. Strukturu *node* proširujemo varijablama *done*, *number* i *origin*. Varijabla *done* poprima vrijednost *true* ili $false$, ovisno o tome je li čvor obilježen ili nije, dok *number* i *origin* predstavljaju redom ime dotičnog čvora (vrha) i vrh iz kojeg se (u dobivenom putu iz Dijkstrinog algoritma) dolazi do dotičnog čvora (vrha). Prvo ubacujemo sve vrhove u F-hrpu *FibHeap*, a pojedini vrh u bilo kojem trenutku algoritma može biti u jednom od tri stanja: obilježen, neobilježen, obrađen. Vrhovi su na početku neobilježeni ($done = false$), a oznake su im postavljene na ∞. Izuzetak tome je istaknuti vrh s koji je obilježen ($done = true$), a oznaka mu je neki konačan broj, npr. 0. Sljedeći korak ponavljamo sve dok imamo vrhova kod kojih $done = true$: iz skupa obilježenih vrhova odaberi onaj vrh v čija je oznaka minimalna i smatraj ga obrađenim, a nakon toga prođi skupom svih bridova kojima je jedan kraj u v te za drugi kraj tog brida w ažuriraj njegovu oznaku u $v.key + length((v, w))$ ako $v.key + length((v, w)) < w.key$ i postavi $w.done = true$. Složenost Dijkstrinog algoritma upravo dolazi iz implementacije upravo opisanog postupka koji se na prvi pogled čini kompliciranim. No, ono što znatno olakšava stvari jest to što znamo koliko ćemo točno puta ponavljati taj postupak. Naime, kako u svakom koraku obradimo jedan čvor, odnosno pronađemo duljinu najjeftinijeg puta do njega iz s, onda nam on više nije potreban nakon što smo ažurirali puteve do njegovih susjeda. Kako je on po konstrukciji algoritma čvor s najmanjom vrijednosti oznake, izbacujemo ga operacijom *deleteMin*.

Da bismo u konačnici obuhvatili sve vrhove, potrebno je provesti n puta gore opisani postupak. Za to nam služi for petlja koja se izvršava n puta, a u svakom prolazu ažuriramo vrijednosti oznaka trenutno "dohvatljivih" vrhova uz pomoć operacije $decreaseKey$ te potom izbacujemo vrh s minimalnom oznakom. Vrijedi napomenuti da upravo $while$ petlja ažurira cijene bridova, no način na koji smo reprezentirali graf garantira da se operacija $decreaseKey$ neće izvršiti više od m puta. Sažmimo ukratko algoritam radi ispitivanja složenosti: ubacujemo n čvorova operacijom $insert$, n puta izvodimo operaciju $deleteMin$, dok se operacija $decreaseKey$ u najgorem slučaju poziva m puta. Iz toga zaključujemo da je složenost algoritma:

$$O(n + n \cdot \log n + m) = O(n \cdot \log n + m).$$

Važno je napomenuti kako prikaz grafa može utjecati na složenost algoritma. Klasična matrica susjedstva bi pogoršala njegovo izvršavanje jer bi za pojedini vrh morali sekvencijalno čitati cijeli redak matrice kako bismo pronašli njegove susjede. Gore dobivena ocjena složenosti ipak se može ostvariti korištenjem matrice susjedstva, no umjesto jedne, trebat će nam dvije takve matrice. Prva matrica je dana sa $struct\ node^*graph$, a njezino značenje je jednostavno. Element $graph[i][j]$ je pokazivač na susjeda vrha s imenom i. Ono što je bitno jest da se ta matrica popunjava s lijeva na desno, tj. ako vrh s imenom i ima k susjeda, onda su $graph[i][0], ..., graph[i][k-1]$ popunjeni odgovarajućim adresama, dok je ostatak polja inicijalizirano na $NULL$. Time efikasno pronalazimo sve susjede određenog vrha. No, nemamo nikakvu informaciju o cijeni brida (i, j). To rješavamo drugom matricom susjedstva $int\ price$ gdje element $price[i][j]$ predstavlja vrijednost (cijenu) brida (i, j). Naravno, da bismo mogli reprezentirati cijeli graf, dovoljno je da obje matrice budu dimenzije $n \times n$. Ovom alternativom je i dalje korišten prostor za spremanje grafa reda veličine n^2, kao što je to slučaj kod standardne matrice susjedstva. Nešto kompaktnija alternativa izložena je u Primovom algoritmu, a koristi se vezana lista susjedstva.

5.3 Primov algoritam

Dan je neusmjereni graf $G = (V, E)$ pri čemu V predstavlja skup vrhova, a E skup bridova te je svakom bridu pridružena njegova vrijednost. Dodatno, definiramo $n \coloneqq |V|$ i $m \coloneqq |E|$. Primov algoritam pronalazi njegov podgraf $G^{'} = (V, E^{'})$ takav da je $E^{'} \subseteq E$ i zbroj vrijednosti bridova u $E^{'}$ je najmanji mogući. Iz definicije problema je jasno da $G^{'}$ mora biti stablo koje nazivamo *minimalno razapinjuće stablo.*
Na početku se stvara prazno stablo, odnosno njegov (prazan) skup vrhova

T u kojega se dodaje proizvoljan vrh grafa G. Nakon toga se bira brid s najmanjom vrijednosti, ali pod uvjetom da mu je jedan kraj vrh u T, a drugi u $V \backslash T$. Nakon odabira brida, vrh iz $V \backslash T$ ubacujemo u skup T. Taj se postupak ponavlja sve dok $T \neq V$. Fibonaccijeva hrpa se koristi tako da se na početku svi vrhovi grafa G ubace u F-hrpu *FibHeap* s vrijednošću oznake ∞, osim početnog vrha kojem je vrijednost oznake $-\infty$. Zatim se uz pomoć *decreaseKey* svim vrhovima koji nisu u T vrijednost oznake postavlja na najmanju vrijednost brida koji ih spaja s nekim vrhom iz T. Potom se poziva funkcija *deleteMin* kako bi samo vrhovi koji još nisu u T ostali u *FibHeap*. Konačno, čvoru s minimalnom konačnom vrijednosti oznake postavljamo oznaku na $-\infty$. Ovdje uočavamo da s oznakom $-\infty$ obilježavamo vrhove koji su u T pa na to moramo dodatno paziti kada provjeravamo susjede nekog vrha. Prethodni postupak se ponavlja n puta. Dodatno, strukturu *node* smo nadogradili varijablama *vertex* i *antecedent* koje predstavljaju brid čiji je jedan kraj dotični *node* s imenom *vertex*, a drugi s imenom *antecedent*. To nam je potrebno kako bismo u tijeku algoritma mogli ispisati bridove (zajedno s njihovim vrijednostima) koji čine minimalno razapinjuće stablo. Između ostalog, nadogradili smo i funkciju *insert* dodatnim argumentom koji služi za ime (*vertex*) čvora. Dakle, za cijeli algoritam potrebno je n operacija *insert* za dodavanje svih vrhova u Fibonaccijevu hrpu, n operacija $deleteMin$ i najviše m operacija $decreaseKey$. Stoga je složenost jednaka

$$O(n \cdot 1 + n \cdot \log n + m \cdot 1) = O(n \cdot \log n + m).$$

U implementaciji je za prikaz susjeda pojedinog vrha korištena vezana lista susjedstva pa je stoga graf prikazan kao struktura s brojem vrhova i nizom vezanih listi pri čemu svaki element niza $array[i]$ predstavlja pokazivač na prvog susjeda vrha s imenom i. Standardan zapis grafa preko matrice susjedstva je znatno jednostavniji, no pogoršao bi vrijeme izvršavanja algoritma. Eventualna alternativa koja i dalje koristi svojevrsnu matricu susjedstva može se vidjeti u reprezentaciji grafa u Dijkstrinom algoritmu.
Postoji unaprjeđenje Primovog algoritma uz korištenje Fibonaccijeve hrpe čija modifikacija daje još bolju ogradu za pronalazak minimalnog razapinjućeg stabla. Više o tome može se pronaći u [5].

6 Zaključak

Fibonaccijeva hrpa predstavlja jednu efikasnu implementaciju prioritetnog reda koja je prvobitno razvijena radi ubrzanja algoritama na grafovima. Osim toga, svoju primjenu pronalazi i u algoritmima selektiranja, simulacijama temeljenim na događajima te mnogim drugim problemima srodnim pronalasku najkraćeg puta. Unatoč izvrsnim vremenima izvršavanja operacija, njezina primjena u praksi nije nikad zaživjela. Razlog tome mogao se uočiti u analizi algoritma sortiranja hrpom - velik broj skrivenih konstanti koje usporavaju izvršavanje operacija. Osim toga, implementacija je zbog dvostrukih cirkularnih vezanih listi dosta složenija. Ovo jasno pokazuje da ako neki algoritam ili struktura ima asimptotski efikasno vrijeme izvršavanja, to ne znači da će biti brzi i u praksi. No, Fibonaccijeva hrpa je jedan lijep primjer strukture podataka koja sadrži visok stupanj interakcije s matematikom, kako u oblikovanju (korištenje korolara 2 za duljinu polja *array* u funkciji *consolidate*), tako i u analizi (uska veza s Fibonaccijevim brojevima) algoritma.

Literatura

[1] Fibonacci heap (Introduction), https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/, 14.12.2022.

[2] Fibonacci heap (Deletion, Extract min and Decrease key), https://www.geeksforgeeks.org/fibonacci-heap-deletion-extract-min-and-decrease-key/, 17.12.2022.

[3] Fibonacci heap (Insertion and Union), https://www.geeksforgeeks.org/fibonacci-heap-insertion-and-union/, 14.12.2022.

[4] Doubly Circular Linked List, https://www.geeksforgeeks.org/insertion-in-doubly-circular-linked-list/, 14.12.2022.

[5] Marino Lončar, Fibonaccijeva hrpa, https://repozitorij.pmf.unizg.hr/islandora/object/pmf%3A5649/datastream/PDF/view, Zagreb, 2016.

[6] Mislav Žanić, Efikasne implementacije prioritetnog reda, https://repozitorij.pmf.unizg.hr/islandora/object/pmf%3A10743/datastream/PDF/view, Zagreb, 2022.

[7] Amortized Analysis (Introduction), https://www.youtube.com/watch?v=WIabUqbOFP0&list=LL&index=11&ab_channel=UzairJavedAkhtar, 17.12.2022.

[8] Amortized Analysis (Aggregate Method), https://www.youtube.com/watch?v=56jy_c6X9uA&ab_channel=UzairJavedAkhtar, 17.12.2022.

[9] Amortized Analysis (Accounting Method), https://www.youtube.com/watch?v=-bHJvg2Jgc&ab_channel=UzairJavedAkhtar, 17.12. 2022.

[10] Amortized Analysis (Potential Method), https://www.youtube.com/watch?v=-xO9TkpMlhs&ab_channel=UzairJavedAkhtar, 17.12.2022.

[11] Robert Manger, Stabla, http://web.studenti.math.pmf.unizg.hr/ manger/spa/SPA-3.pdf, Zagreb, 2022.