UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Svetlana Nikić

Konična drevesa

Delo diplomskega seminarja

Mentor: izred. prof. dr. Andrej Bauer

Ljubljana, 2012

Kazalo

1. Uvod 4
2. Definicija koničnega drevesa 4
3. Monoidi in redukcije 6
3.1. Monoidi 6
3.2. Redukcije 7
4. Redukcije na koničnih drevesih 8
4.1. Redukcija vozlišč 8
4.2. Redukcija koničnega drevesa 9
5. Osnovne operacije 9
5.1. Dodajanje elementa na koncih 9
5.2. Odstranjevanje skrajnih elementov 11
5.3. Združevanje dreves 13
6. Iskanje elementa 14
6.1. Delitev drevesa 15
7. Uporaba 19
7.1. Seznam z naključnim dostopom 20
7.2. Max-prioritetna vrsta 20
7.3. Urejen seznam 21
7.4. Intervalno drevo 21
Literatura 23

Konična drevesa

Povzetek

Konično drevo, ki ga razvijemo iz 2-3 drevesa, je obstojna podatkovna struktura, namenjena predvsem funkcionalnemu programiranju. S koničnim drevesom smo dobili strukturo, ki dostopa do podatkov na obeh koncih v amortizirano konstantnem času, medtem ko tudi ostale osnovne operacije, kot je združevanje dveh dreves in delitev drevesa, dosegajo logaritemsko časovno zahtevnost. Za razliko od drugih struktur, ki so pred tem že imele take časovne zahtevnosti, pa so konična drevesa še vedno dosti preprostejša, tako kot tudi operacije na njih. Dobili smo strukturo, s katero lahko implementiramo mnogo drugih struktur, na primer seznam s slučajnim dostopom, urejen seznam, prioritetno vrsto, iskalno drevo in druge. Torej je konično drevo zelo uporabna obstojna struktura, ki z razvojem paralelnega računanja dobiva vedno večjo vlogo v programerskem svetu.

Finger Trees

Abstract

Finger tree, which we develop from 2-3 trees, is a persistent data structure intended primarily for functional programming. It is a structure that supports access to both ends in constant amortized time, while other basic operations, such as concatenation and splitting, achieve logarithmic bounds. Unlike other structures that have previously achieved these bounds, finger trees are much simpler, as are operations on them. We developed a structure which can implement many other structures, such as random access sequence, ordered sequence, priority queue, search tree and others. Finger tree is a very useful persistent structure which is becoming more important with the development of parallel computing.

Math. Subj. Class. (2010): 68P05, 68N18

Ključne besede: konična drevesa, podatkovne strukture, funkcijsko programiranje

Keywords: finger trees, data structures, functional programming

1. Uvod

Z razvojem večjedrnih procesorjev so le-ti postali vedno bolj uporabni v tehnološkem svetu in tudi v našem vsakdanu. V svetu programiranja se tako čedalje bolj uveljavlja čisto funkcijsko programiranje, ki zna zelo dobro izkoriščati lastnosti večjedrnih procesorjev. Čeprav poznamo funkcijsko programiranje že dalj časa, pa do sedaj ni bilo večje potrebe po implementaciji raznih podatkovnih struktur. Z razvojem tehnologije pa se spreminja tudi to. Ena od podatkovnih struktur, ki je kot nalašč narejena za tako programiranje, je konično drevo. Ta struktura omogoča hiter dostop do podatkov, obenem pa je od svojih predhodnikov dosti bolj preprosta.

Konično drevo je čisto funkcijska podatkovna struktura. Podatkovna struktura je *čisto funkcijska*, če ostane nespremenjena ne glede na uporabo operacij na njej. To pomeni, da operacije, ki jih izvedemo na koničnem drevesu, le-tega ne spremenijo, ampak vedno vrnejo novo različico. To lastnost lahko dobro izkoriščamo s funkcijskim programiranjem, z dobro implementacijo pa lahko zelo izboljšamo tudi prostorsko zahtevnost.

Veliko vlogo pri takih podatkovnih strukturah igra tudi tako imenovano leno vrednotenje. Ko govorimo o vrednotenju, večina funkcijskih programskih jezikov uporablja eno od dveh splošnih strategij vrednotenja, *neučakano* ali *leno*. Rečemo, da jezik uporablja neučakano vrednotenje, če ovrednoti argumente funkcije, preden uporabi na njih funkcijo. Na drugi strani pa pri lenem vrednotenju argumente ovrednoti sproti in samo, če potrebuje rezultat za nadaljnje računanje.

Za implementacijo koničnega drevesa se v splošnem uporablja enega od čisto funkcijskih programskih jezikov, ki uporabljajo leno vrednotenje, na primer Haskell. Namesto o časovni zahtevnosti v najslabšem možnem primeru je torej boljše govoriti o *amortizirani* analizi časovne zahtevnosti. To pomeni, da nas ne zanima zahtevnost ene same ponovitve operacije, ampak kakšno zahtevnost bi imela ena ponovitev operacije v povprečju, če bi jo ponovili velikokrat. Torej dovolimo, da ima lahko operacija v najslabšem primeru slabo časovno zahtevnost, če se le zgodi dovolj redko, da ostale ponovitve operacije s svojimi dobrimi časovnimi zahtevnostmi izboljšajo skupno zahtevnost in s tem povprečno zahtevnost ene same operacije.

Čeprav se v praksi na koničnih drevesih uporablja leno vrednotenje, pa bomo mi večinoma delali neučakano analizo zaradi boljše predstave o operacijah.

2. Definicija koničnega drevesa

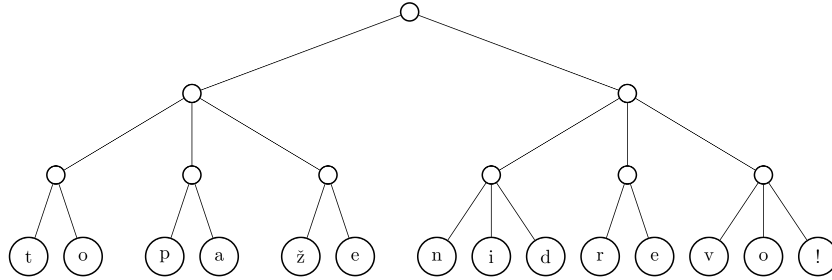
Poglejmo si, kako se je razvilo konično drevo. Za osnovo vzamemo 2-3 drevo, ki mu bomo dodajali lastnosti, dokler ne dobimo koničnega drevesa.

Definicija 2.1. *2-3 drevo* je popolnoma poravnano drevo z lastnostmi:

(1) Vsako notranje vozlišče ima dva sinova in en ključ ali pa tri sinove in dva ključa.
(2) Vsi podatki so shranjeni v listih drevesa.
(3) Vsi listi so na istem nivoju.

Po tej definiciji, ki je vzeta iz knjige [4], 2-3 drevo shranjuje v notranjih vozliščih ključe, ki pa jih mi v osnovni definiciji koničnega drevesa ne bomo rabili, zato jih odstranimo.

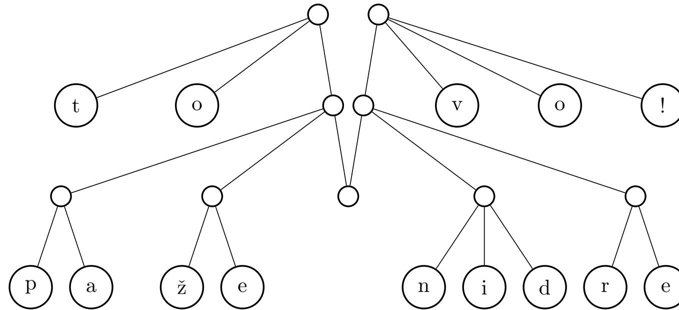
Ena od želenih lastnosti koničnih dreves je dodajanje in odstranjevanje prvega in zadnjega podatka drevesa v konstantnem času. V 2-3 drevesu je časovna zahtevnost



Slika 1. Primer: 2-3 drevo.

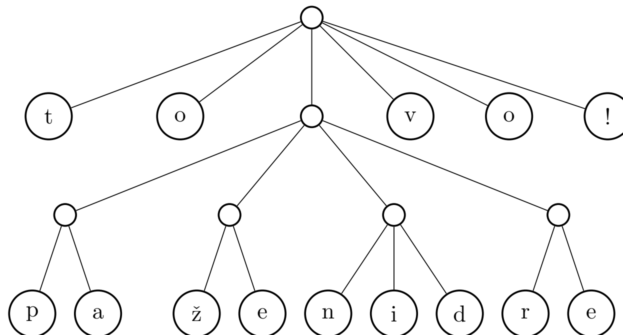
teh operacij logaritemsko odvisna od velikosti drevesa, saj prvi in zadnji podatek ležita na globini $O(\log n)$, kjer je n število vseh podatkov.

Da bi preuredili 2-3 drevo v konično drevo, si pomagamo s kazalci. *Kazalec* je struktura, ki zagotavlja učinkovit dostop do vozlišč blizu določene lokacije. Mi želimo imeti hiter dostop do elementov na obeh koncih zaporedja, zato želimo imeti kazalce na obeh koncih drevesa. Tako drevo lahko dobimo, če 2-3 drevo primemo v obeh skrajnih notranjih vozliščih ter ti dve vozlišči dvignemo.



Slika 2. Primer: dvig 2-3 drevesa.

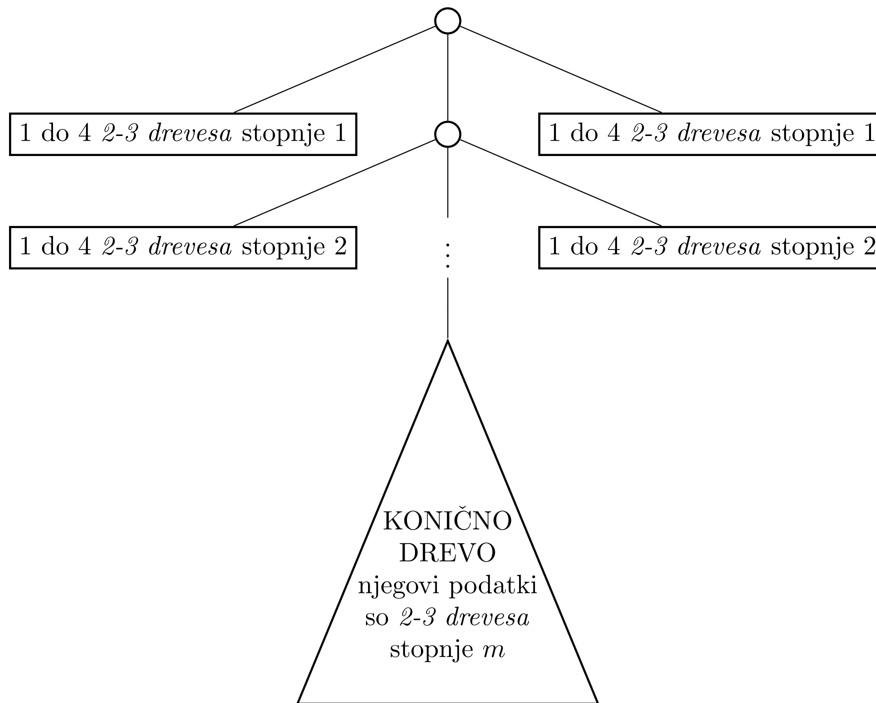
Zaradi lastnosti 2-3 drevesa, da so vsi listi na isti globini, sta leva in desna hrbtenica enako dolgi. Hrbtenici lahko združimo v eno samo hrbtenico, prav tako pa združimo tudi ustrezne pare vozlišč. Glede na hrbtenico oštevilčimo nivoje od vrha navzdol. Na vsakem nivoju imamo na levi in desni strani 2-3 drevesa, razen morda na zadnjem, če je bila stopnja korena začetnega 2-3 drevesa 3. V tem primeru imamo na zadnjem nivoju samo eno 2-3 drevo. Enemu samemu 2-3 drevesu, ne glede na njegovo stopnjo, rečemo *enojec.* En sam element je 2-3 drevo stopnje 1, torej tudi enojec.



Slika 3. Primer: konično drevo.

Na prvem nivoju imamo na vsaki strani dve ali tri drevesa stopnje 1, torej kar same podatke. Na vseh ostalih nivojih (razen morda zadnjem) imamo na vsaki strani eno ali dve drevesi. Na drugem nivoju so ta drevesa stopnje 2, na tretjem stopnje 3 itd. Skupini poddreves na eni strani na enem nivoju rečemo *konica*. Konico predstavimo kot seznam poddreves, pri čemer seveda ohranimo isti vrstni red, kot je v drevesu.

Tako drevo nam da osnovo za konično drevo. Da bi ga čim bolj posplošili in omogočili čim boljše delovanje nekaterih operacij na drevesu, dovolimo, da ima drevo v vsaki konici od enega do štiri poddrevesa.



Slika 4. Splošno konično drevo.

Opomba: na zadnjem nivoju imamo lahko tudi samo eno 2-3 drevo (enojec) stopnje m (m je število nivojev), ki visi na sredini.

3. Monoidi in redukcije

3.1. **Monoidi.** S pomočjo koničnih dreves bomo želeli implementirati mnogo struktur, na primer prioritetno vrsto, iskalno drevo, urejen seznam, itd. Da lahko eno samo drevo uporabimo kot vse te podatkovne strukture, pa nam omogočajo monoidi.

Spomnimo se definicije monoida (vzeta iz [3]):

Definicija 3.1. Naj bo $\circ : M \times M \to M$ binarna operacija na M. Potem rečemo, da je $(M, \circ)$ *polgrupa*, če je operacija $\circ$ asociativna.

Če je $(M, \circ)$ polgrupa in v njej obstaja enota e, potem rečemo, da je $(M, \circ)$ *monoid*.

Če predstavimo zaporedje n elementov s koničnim drevesom, lahko pokažemo, da imamo dostop do kateregakoli elementa zaporedja v času $O(\log n)$, pri čemer leži element, ki je od bližjega konca oddaljen za d, na globini $O(\log d)$. Vidimo, da imamo največjo časovno zahtevnost, ko želimo dostopati do elementa, ki leži blizu sredine zaporedja, saj leži v koničnem drevesu na zadnjem nivoju. Teh nivojev

je $O(\log n)$. Na zadnjem nivoju so 2-3 drevesa globine $O(\log n)$, torej da pridemo do elementa na dnu tega drevesa, potrebujemo vse skupaj $O(2 \log n)$, kar je enako $O(\log n)$.

Poglejmo si dva primera. V prvem primeru imamo zaporedje in iščemo njegov prvi element. V drugem pa imamo prioritetno vrsto iz katere želimo izbrisati element. Čeprav sta primera na prvi pogled zelo različna, pa se ju da rešiti na zelo podoben način ravno z uporabo monoidov. Več o samem reševanju v poglavju 7.

3.2. **Redukcije.** *Redukcija* je funkcija, ki reducira strukturo podatkovnega tipa $f(a)$ v eno samo vrednost tipa a.

Redukcija prazne strukture vrne konstanto, na primer 0, vmesne rezultate pa združi z binarno operacijo $\oplus$. Na začetku podamo še neko začetno vrednost.

Primer 3.2. Recimo, da vzamemo za tip a število, za $f(a)$ pa seznam števil. Torej bi seznam števil z redukcijo reducirali v eno samo število, pri čemer moramo definirati še operacijo $\oplus$, s katero bomo reducirali seznam.

Če bi vzeli za $\oplus$ seštevanje, bi seznam $[5, 3, 2, 8]$ z začetno vrednostjo 0 reducirali v $0 + 5 + 3 + 2 + 8 = 18$.

Če pa bi vzeli za $\oplus$ funkcijo maksimum, bi ta isti seznam reducirali v 8. Za začetno vrednost lahko vzamemo na primer minus neskončno ali pa kar prvi element seznama.

V primeru, da za $\oplus$ vzamemo odštevanje, pa je rezultat odvisen od tega, kako gnezdimo funkcijo. Recimo, da je začetna vrednost enaka 0. Rezultat je lahko $(((0 - 5) - 3) - 2) - 8 = -18$, če vedno odštevamo od levega elementa, lahko je $5 - (3 - (2 - (8 - 0))) = -4$, če odštevamo z druge strani ali pa recimo $5 - (((3 - 0) - 2) - 8) = 12$ z nekim mešanim gnezdenjem.

Če redukcije delamo na monoidih, dobimo vedno enak rezultat, ne glede na gnezdenje operacije $\oplus$. Za poljubno konstanto in poljubno binarno operacijo pa je treba definirati redukcije za vsak primer gnezdenja posebej.

Če gnezdimo samo v desno ali samo v levo, smo dobili dve posebni gnezdenji, ki ju poimenujemo *usmerjeni redukciji*. Usmerjeni redukciji tako veljata za poljubno konstanto in binarno operacijo, pri čemer ima lahko operacija za argumente tudi različne tipe elementov.

Usmerjena redukcija *reducirajl* gnezdi v levo. Vzame torej funkcijo, ki iz prvega argumenta tipa a in drugega tipa b vrne nekaj tipa b, in naredi novo funkcijo, ki za prvi argument vzame element tipa $f(a)$. Drugi argument in končni rezultat ostaneta enakega tipa kot pri začetni funkciji, torej tipa b. Drugi argument ima predvsem vlogo začetne vrednosti oziroma vmesnega rezultata.

V zadnjem delu primera 3.2 bi leva redukcija vzela za funkcijo običajno, levo gnezdeno odštevanje (oba argumenta in rezultat so števila) in vrnila funkcijo, ki za prvi argument vzame seznam števil in za drugi argument število, ter vrnila novo število.

Podobno *reducirajd*, ki gnezdi v desno, vzame funkcijo, ki ima za prvi argument element tipa b, za drugega pa tipa a in vrne nekaj tipa b. Redukcija nato vrne funkcijo, ki za drugi argument namesto elementa tipa a vzame element tipa $f(a)$ in še vedno vrne element tipa b, pri čemer ostane tudi prvi argument tipa b.

Na redukcije pa lahko gledamo tudi z druge strani in sicer namesto, da govorimo o redukciji funkcije tipa $f(a)$ na vrednost tipa a, lahko govorimo o dvigu operacije na drugo strukturo.

Torej če imamo element tipa a, ki ga operacija $\prec$ preslika v element tipa b, bo $reducirajd(\prec)$ preslikala celo strukturo tipa $f(a)$ v element tipa b.

V primeru 3.2 torej namesto, da rečemo, da seznam reduciramo na število, lahko rečemo, da smo funkcijo, ki preslika število v število, dvignili tako, da preslika cel seznam v število.

Primer 3.3. Napisati si želimo funkcijo *vSeznam*, ki bo neko strukturo s spremenila v seznam.

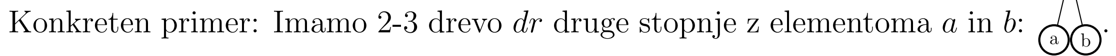
Pomagamo si z redukcijami in dvema vgrajenima funkcijama jezika Haskell, *foldr* in *foldl* [1].

foldr vzame tri argumente: binarno funkcijo, začetno vrednost in podatkovno strukturo. Elemente podatkovne strukture nato postopoma s funkcijo z leve dodaja začetni vrednosti.

foldl je simetrična funkcija.

Redukcijo *reducirajd*, ki bo za argumente dobila neko operacijo, ki jo označimo z $\prec$, začetno vrednost i in strukturo s, definiramo kot $foldr \prec i\ s$.

Definirajmo funkcijo *dodajVSeznam*, ki samo doda podan element z leve v podan seznam. Z $dodajVSeznam'$ označimo funkcijo $reducirajd(dodajVSeznam)$. Če sedaj to redukcijo izvedemo na strukturi s in praznem seznamu (začetna vrednost i), bomo spremenili strukturo s v seznam. Elementi seznama bodo elementi strukture s. To je ravno naša iskana funkcija *vSeznam*.

Konkreten primer: Imamo 2-3 drevo *dr* druge stopnje z elementoma a in b: .

vSeznam(*dr*)=*foldr*(*dodajVSeznam*, [], (a)(b))=
=*dodajVSeznam*((a), *foldr*(*dodajVSeznam*, [], (b)))=
=*dodajVSeznam*((a), *dodajVSeznam*((b) , [ ]))=
=*dodajVSeznam*((a), [(b)])=
=[(a),(b)]

Opomba: Vse kode v tem delu so prirejena različica Haskell kode za boljše razumevanje.

4. Redukcije na koničnih drevesih

Pri koničnih drevesih se pojavita dve osnovni redukciji, to sta redukcija vozlišč in redukcija celega koničnega drevesa. Obe nam bosta prišli prav pri analizi operacij na drevesih.

4.1. Redukcija vozlišč.

Pri redukciji vozlišč moramo ločiti primera, če ima vozlišče dva ali tri sinove. V obeh primerih samo apliciramo funkcijo glede na to, v katero smer je usmerjena redukcija.

Poglejmo si *reducirajd*, ki sprejme za argumente neko funkcijo $\prec$, vozlišče z dvema sinovoma in strukturo s. Vozlišče ima dva sinova in *reducirajd* bo strukturi s najprej s funkcijo $\prec$ dodala drugega sina, nato pa še prvega. Tako predelana struktura bo rezultat *reducirajd*. Podobno bi v primeru, da ima vozlišče tri sinove, strukturi najprej z leve dodali zadnjega, nato drugega in na koncu še prvega.

Ravno obratno deluje *reducirajl*, ki ima za argument funkcijo $\succ$. Kot rezultat bomo dobili strukturo, ki ji z desne dodamo najprej prvega sina, nato pa še drugega. Če ima vozlišče tri sinove, dodamo nazadnje še tretjega.

4.2. **Redukcija koničnega drevesa.** Poglejmo si najprej desno usmerjeno redukcijo drevesa (dodajamo vedno v desno smer). *reducirajd* bo za prvi argument vedno dobila neko funkcijo $\prec$ ter za tretjega strukturo s. Tako kot pri redukciji vozlišč sedaj ločimo primere glede na to, kaj je drugi argument. Če je drugi argument prazno drevo, bo *reducirajd* vrnila nespremenjen s. V primeru, da za drugi argument dobi enojec x, torej eno samo 2-3 drevo, vrne strukturo s, ki ji z leve dodamo s funkcijo $\prec$ enojec x. Na prvem nivoju sta ta enojca 2-3 drevesi prve stopnje, torej samo elementa. Redukcija pa se lahko pokliče tudi globje v drevesu, kjer sta enojca 2-3 drevesi višjih stopenj.

Ostane nam še primer, ko je drugi argument globoko drevo. Uvedimo konstruktor *Globoko*, ki dobi kot argumente *predpono* (leva konica na prvem nivoju), *pripono* (desna konica) in *vmesno drevo* (vse ostalo kar potrebujemo za sestavo koničnega drevesa) ter zna iz njih narediti globoko konično drevo. Zato lahko vsako globoko konično drevo zapišemo s temi tremi deli. Najprej s funkcijo $\prec'$ poimenujemo *reducirajd*$(\prec)$, s funkcijo $\prec''$ pa poimenujemo *reducirajd*(*reducirajd*$(\prec)$). Naša funkcija *reducirajd* bo na globokem drevesu vrnila strukturo s, ki ji najprej z leve z $\prec'$ dodamo pripono, dobljeni strukturi nato z leve z $\prec''$ dodamo vmesno drevo, nato pa z leve dodamo z $\prec'$ še predpono.

Simetrično deluje leva redukcija *reducirajl*.

5. Osnovne operacije

5.1. **Dodajanje elementa na koncih.** V tem delu bomo pokazali, da lahko dodamo element koničnemu drevesu z levega ali z desnega konca v konstantnem času. Če dodajamo z leve, bomo operacijo označili z $\triangleleft$. Če dodajamo z desne, pa jo bomo označili z $\triangleright$.

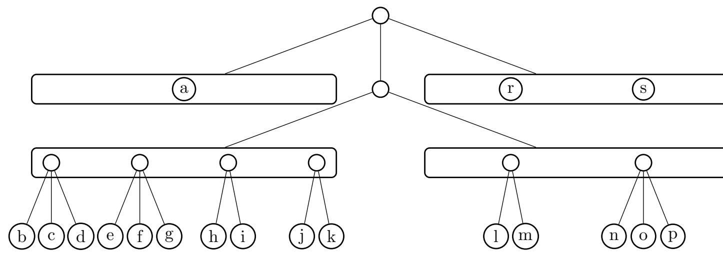
Poglejmo si primer, ko element dodajamo z leve. Najprej predelajmo osnovna primera. Če element a dodajamo praznemu drevesu, bomo dobili enojec a in če dodajamo element a enojcu b, bomo dobili globoko drevo, ki ima v predponi a in v priponi b. Dodajanje se lahko pokliče globje v drevesu, torej sta v tem primeru element a in enojec, ki mu a dodajamo, 2-3 drevesi višje stopnje, in sicer bosta na i-tem nivoju stopnje i.

Ostane nam še dodajanje elementa globokemu drevesu. Drevo si zopet predstavljamo v treh delih: predponi, priponi in vmesnem drevesu m. Drevesu dodajamo element z leve, torej ga dodajamo v predpono. Edina omejitev drevesa, na katero moramo paziti, je ta, da imamo v konici lahko največ štiri elemente. Če imamo torej drevo, ki ima v konici enega, dva ali tri elemente, in mu želimo z leve dodati še en element, element preprosto dodamo in smo še vedno znotraj omejitev. Poglejmo pa si primer, ko dodamo element drevesu, ki ima v predponi že štiri elemente. V konici imamo sedaj pet elementov, od katerih prva dva pustimo tako kot sta, ostale tri pa vstavimo v novo 2-3 drevo prve stopnje s tremi sinovi. To drevo zdaj dodajamo vmesnemu drevesu m z rekurzivnim klicem naše metode.

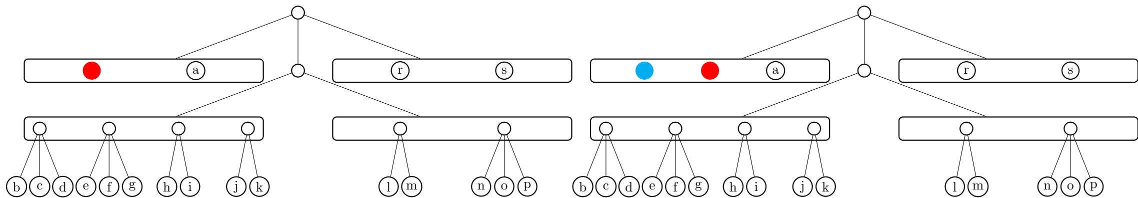
Operacijo bi lahko definirali tudi tako, da bi v primeru, ko imamo v predponi pet elementov, pustili v konici tri elemente in na nižji nivo dali samo dva.

Dodajanje elementa z desne je simetrično.

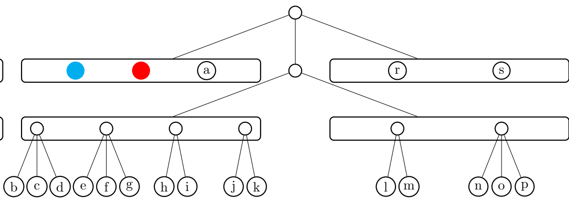
Primer 5.1. Na naslednji strani imamo primer začetnega drevesa, na katerem si bomo ogledali dodajanje elementov. Drevo ima v predponi samo en element, zato prvo, drugo in tretje dodajanje drevesa ne spremenijo kaj dosti. Na primeru na koncu tudi vidimo, kako dodajamo element drevesu, ki ima v predponi že štiri elemente.



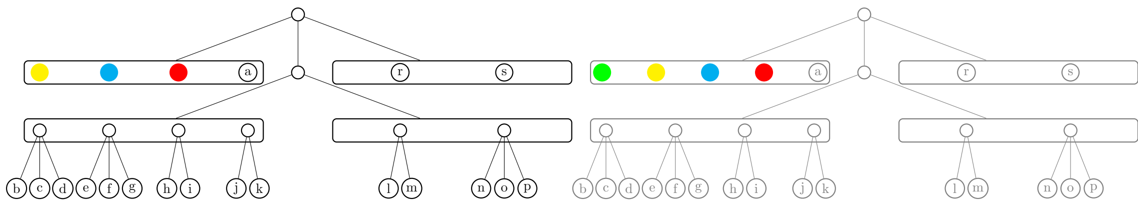
(A) Začetno drevo.



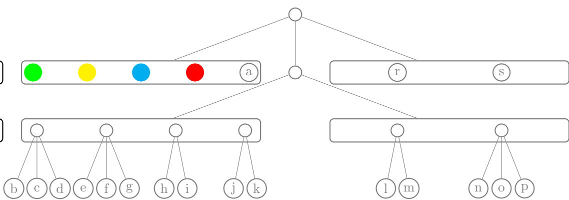
(B) Dodali smo rdeč element in dobili konično drevo.



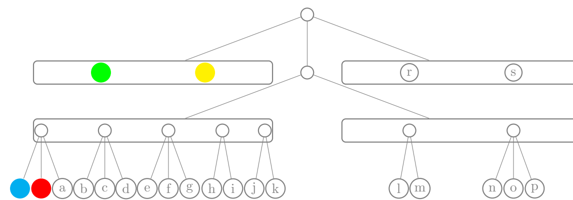
(C) Dodali smo še moder element.



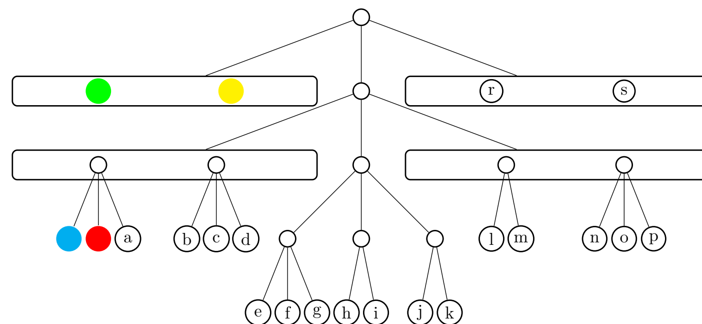
(D) Dodali smo rumen element. Sedaj imamo v predponi koničnega drevesa štiri elemente.



(E) Dodali smo še zelen element. V predponi je sedaj pet elementov, zato to ni konično drevo! Drevo je treba preurediti.



(F) Prvi korak preurejanja smo naredili, a to še ni konično drevo.



(G) Dobili smo konično drevo in tako uspešno dodali tudi zelen element v drevo.

SLIKA 5. Dodajanje elementov z leve.

Operacija dodajanja bi se lahko rekurzivno klicala vse do dna drevesa in bi tako imela časovno zahtevnost v najslabšem primeru $O(\log n)$. Pokazali bomo, da je amortizirana časovna zahtevnost konstantna.

Operacija na nižji nivo napreduje samo, če je naletela na konico s štirimi elementi. Taki konici rečemo *nevarna* konica. Istočasno, ko je operacija napredovala nivo nižje, je spremenila to konico v konico z dvema elementoma. Ker s take konice operacija ne more napredovati, ji rečemo *varna*. Torej bosta naslednji dve operaciji neleteli na varni konici (tudi konica s tremi elementi je varna).

Na prvem nivoju se izvede vseh N operacij, na drugi nivo pa napreduje največ tretjina operacij. Na tretjega napreduje največ devetina itd. Vse skupaj bomo torej naredili največ $N + \frac{N}{3} + \frac{N}{9} + \ldots$ operacij. To pomeni, da če izvedemo operacijo N-krat, pri čemer je N veliko število, bomo potrebovali za izvedbo ene operacije približno $\frac{N+\frac{N}{3}+\frac{N}{9}+\ldots}{N} = 1 + \frac{1}{3} + \frac{1}{9} + \cdots = \frac{3}{2}$ operacij.

Povprečno število operacij je konstantno, torej je amortizirana časovna zahtevnost res konstantna.

5.2. **Odstranjevanje skrajnih elementov.** Pri odstranjevanju elementa z ene strani imamo podoben problem kot pri dodajanju. Tokrat lahko element odstranimo brez problema, če ima konica dva, tri ali štiri elemente, če pa ima samo enega, konica po odstranjevanju ostane prazna. V tem primeru z nivoja nižje odstranimo prvo 2-3 drevo, vzamemo njegova dva ali tri elemente in ju postavimo v našo prazno konico. Lahko se zgodi, da bo sedaj konica na drugem nivoju prazna, ampak to spet rešimo z odstranjevanjem drevesa na tretjem nivoju itd., če je to potrebno.

Če pridemo do praznega vmesnega drevesa, naslednje drevo odstranimo s pripone, s tem da se tu rekurzija ustavi. Če je pripona imela samo eno drevo, to postane enojec na tem nivoju, če pa jih je imela več, zadnje ostane v priponi, ostala pa gredo v predpono. Koliko dreves gre v predpono variira glede na definicijo operacije, ki seznam dreves pretvori v drevo.

Idejo dodajanja elementa se da s funkcijskim programskim jezikom zapisati tako, kot smo povedali, medtem ko se je treba formalnega zapisa odstranjevanja elementa lotiti malo drugače. Podrobno si poglejmo samo odstranjevanje elementa z leve.

Najprej definiramo operacijo *vDrevo*, ki jo bomo potrebovali kasneje. *vDrevo* definiramo s pomočjo $\lhd'$ in $\rhd'$, ki sta dviga operacij $\lhd$ in $\rhd$, torej $\lhd' = \mathit{reducirajd}(\lhd)$ ter $\rhd' = \mathit{reducirajl}(\rhd)$. Vse kar sedaj *vDrevo* na strukturi s naredi, je, da praznemu drevesu z operacijo $\lhd'$ z leve doda elemente s.

Sedaj definiramo dve novi funkciji, ki se lahko kličeta med seboj: pogled_L in $\mathit{globoko}_L$.

pogled_L je funkcija, ki vrne Nic_L, če je drevo prazno. Če je drevo, na katerem izvedemo pogled_L, enojec x, vrne funkcija enojec x in prazno drevo. Ta funkcija se lahko kliče globje v drevesu, torej je enojec spet 2-3 drevo, katerega stopnja je odvisna od nivoja, na katerem se nahajamo. Če pa je drevo globoko (ima predpono, vmesno drevo m in pripono), funkcija vrne prvo drevo predpone in s funkcijo $\mathit{globoko}_L$ skonstruira novo drevo iz preostanka predpone, vmesnega drevesa m in pripone.

Funkcija $\mathit{globoko}_L$ dobi tri argumente: predpono, vmesno drevo m in pripono, pri čemer je predpona lahko prazna. V primeru, da je predpona prazna, izvede funkcijo pogled_L na m. Če ta vrne Nic_L, potem funkcija $\mathit{globoko}_L$ vrne pripono, ki jo še prej s funkcijo *vDrevo* spremeni, kot ime pravi, v drevo. Če pa funkcija pogled_L vrne element (2-3 drevo), recimo mu a in novo konično drevo brez tega elementa, recimo drevesu m', potem funkcija $\mathit{globoko}_L$ vrne drevo, ki ima za predpono s funkcijo

vSeznam spremenjen a, za vmesno drevo m' in nespremenjeno pripono. a je 2-3 drevo stopnje i in ga spremenimo v seznam dreves stopnje $i-1$, saj želimo imeti v predponi stopnjo plitvejša drevesa kot pa se nahajajo v vmesnem drevesu.

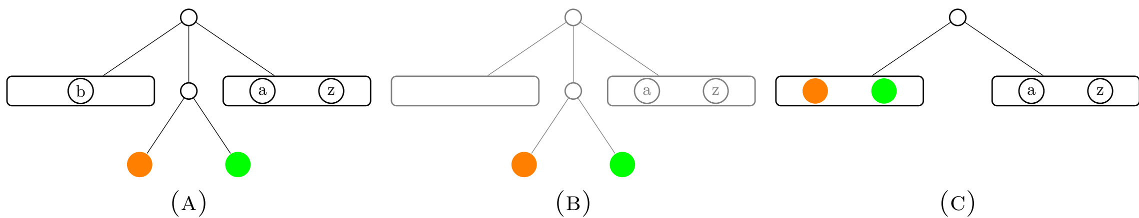
Če predpona drevesa, ki ga je za argument dobila funkcija $globoko_L$, ni prazna, drevo pusti pri miru.

```
globoko_L(predpona, m, pripona) :
    if predpona==[]:
        if pogled_L(m)==Nic_L:  return vDrevo(pripona)
        else (pogled_L(m)==(a, m')):  return Globoko(vSeznam(a), m', pripona)
    else:  return Globoko(predpona, m, pripona)
```

Če sedaj definiramo rep_L kot funkcijo, ki vrne samo drugi del rezultata funkcije $pogled_L$ (v primeru, da $pogled_L$ vrne Nic_L, rep_L ne vrne ničesar), je to konično drevo brez prvega elementa.

Želeli smo odstraniti prvi element iz drevesa in rešitev nam vrne ravno funkcija rep_L.

Primer 5.2. Oglejmo si kratek primer odstranjevanja elementa drevesa, ki ima v predponi samo en element.



Slika 6. Odstranjevanje skrajno levega elementa.

Na sliki (A) imamo začetno drevo, na sliki (B) pa stanje, ko odstranimo element. To ni konično drevo, zato ga je treba preurediti. Preuredimo tako, da dobimo drevo na sliki (C).

Podobno kot rep_L lahko zapišemo funkcijo $glava_L$, ki vrne prvi element.

S tako definirano operacijo $pogled_L$ si lahko pomagamo tudi pri operaciji *jePrazno*, ki pove ali je konično drevo prazno ali ne. Namreč, če $pogled_L$ vrne Nic_L, potem *jePrazno* vrne *True*, sicer vrne *False*.

Časovna zahtevnost se izračuna na enak način kot pri dodajanju elementa. Edina razlika je, da je nevarna konica tista z enim samim elementom. Ta konica se nato spremeni v konico z dvema ali tremi elementi, kar pomeni, da na drug nivo napreduje največ polovica operacij, četrtina na tretji nivo, osmina na naslednji itd. Ko je število ponovitev operacije veliko, nam to zopet prinese amortizirano časovno zahtevnost $O(1)$.

To so časovne zahtevnosti, če vse operacije izvedemo takoj, lahko pa pokažemo, da dosežemo enake časovne zahtevnosti tudi, ko govorimo o časovni zahtevnosti, kjer izkoriščamo leno vrednotenje (uporabljamo predvsem, ko delamo z obstojnimi strukturami). To nam zagotovi, da se transformacije globoko v drevesu ne bodo dogajale, dokler ne bomo rabili rezultata operacije, ki je šla tako globoko v drevo. Zaradi lastnosti varnih in nevarnih konic se bo v tem času zgodilo dovolj operacij na površini, ki bodo poplačale za to eno, drago operacijo.

Formalno se da to pokazati z Okasakijevo debetno analizo [5].

5.3. **Združevanje dreves.** Za identiteto pri združevanju dreves vzamemo prazno drevo. Združevanje dreves, pri čemer je eno od njih enojec, pa je enako kot prej obravnavano dodajanje elementa drevesu. Edini kompleksen primer je združevanje dveh globokih dreves.

Za predpono novega drevesa vzamemo kar predpono prvega, za pripono pa pripono drugega. Iz ostalih delov teh dveh dreves moramo sestaviti novo drevo, ki ima za elemente eno stopnjo globlja 2-3 drevesa. Vidimo, da bo potrebno spreminjati seznam 2-3 dreves v seznam eno stopnjo globljih 2-3 dreves.

Potrebovali bomo že prej definirani funkciji $\lhd'$ in $\rhd'$, ki sta dviga funkcij $\lhd$ in $\rhd$. Poleg tega definiramo še funkcijo *vozlisca*, ki bo pretvorila seznam elementov v seznam 2-3 dreves iz teh elementov. Vhodni seznam bo imel, kot bomo videli, vedno vsaj dva elementa.

Funkcija *vozlisca* bo seznam dveh elementov spremenila v seznam z enim vozliščem, ki ima dva sinova. Podobno bo seznam treh spremenila v seznam z vozliščem, ki ima tri sinove. Če so elementi štirje, bomo dobili dve vozlišči z dvema sinovoma, če pa je vozlišč pet ali več, pa iz prvih treh naredimo vozlišče s tremi sinovi, na preostalih pa spet pokličemo funkcijo *vozlisca*. Seznamu, ki nam ga znova poklicana funkcija vrne, dodamo še prvo vozlišče.

Sedaj lahko definiramo glavno funkcijo *app3*, ki nam bo pomagala pri združevanju dveh dreves. Ta funkcija bo prejela tri argumente: prvo drevo, seznam s in drugo drevo. Seznam s se bo pojavil zaradi združevanja vmesnih konic.

Če je prvo drevo prazno, funkcija z $\lhd'$ doda seznam s z leve drugemu drevesu. Če je prvo drevo enojec, naredi enako, s tem da potem doda še enojec novemu drevesu z leve s funkcijo $\lhd$. Če je prazno drugo drevo, doda s z desne prvemu drevesu s funkcijo $\rhd'$, in če je drugo drevo enojec, z desne doda potem še enojec z $\rhd$. V primeru, da sta obe drevesi globoki, označimo njune dele: $predpona_1$, m_1 in $pripona_1$ so deli prvega drevesa, $predpona_2$, m_2 in $pripona_2$ pa deli drugega drevesa. *app3* sestavi novo drevo, ki ima predpono $predpona_1$, pripono $pripona_2$, za vmesni del pa se pokliče rekurzivno, s tem, da za prvo drevo vzame m_1, za drugo m_2, vmesni seznam pa naredi tako, da združi sezname $pripona_1$, s in $predpona_2$ ter na njih izvede funkcijo *vozlisca*.

Združevanje dreves (označimo z $\bowtie$) dobimo sedaj tako, da samo pokličemo to operacijo s praznim seznamom.

$app3(d_1, s, d_2):$
  if d_1==*Prazno*: **return** $s \lhd' d_2$
  elif d_2==*Prazno*: **return** $d_1 \rhd' s$
  elif d_1==*Enojec* x: **return** $x \lhd (s \lhd' d_2)$
  elif d_2==*Enojec* x: **return** $(d_1 \rhd' s) \lhd x$
  else $(d_1 = (predp_1, m_1, prip_1),\ d_2 = (predp_2, m_2, prip_2))$:
    return $Globoko(predp_1, app3(m_1, vozlisca(prip_1 + s + predp_2), m_2), prip_2)$

$\bowtie (d_1, d_2):$
  return $app3(d_1, [], d_2)$

Vmesni seznam ima vedno največ štiri 2-3 drevesa, torej ima dodajanje konstantno časovno zahtevnost. Združevanje se rekurzivno kliče dokler ne pridemo do dna manjšega drevesa, pri vsakem klicu pa naredimo konstantno število korakov. Skupna časovna zahtevnost je torej $O(\log(\min\{n_1, n_2\}))$.

6. Iskanje elementa

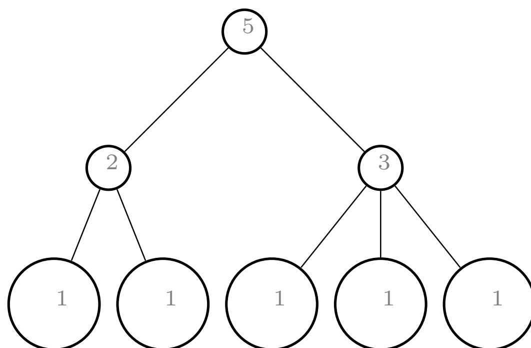
Pri mnogo operacijah, kot so na primer brisanje prvih n elementov, razdelitev drevesa na dve poddrevesi in mnoge druge, je osnovni problem vedno iskanje elementa pod določenimi kriteriji.

Pri tem nam bodo pomagale operacije, ki elementom in drevesom izračunajo oziroma izmerijo neko vrednost. Rekli jim bomo kar *mere*. To so operacije, ki imajo enoto in so asociativne, torej bomo imeli opravka s prej omenjenimi monoidi, saj želimo imeti enake vrednosti mere, ne glede na gnezdenje operacij.

V vsako vozlišče uvedemo predpomnilnik, v katerega bomo shranili vnaprej izračunane podatke.

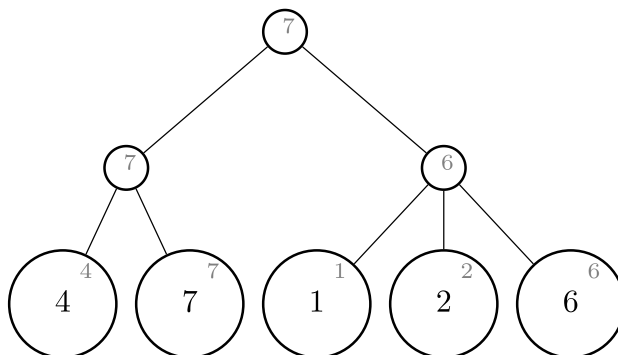
Če listom dodelimo neke vrednosti v predpomnilnik, lahko nato z uporabo neke asociativne operacije izračunamo vrednosti tudi v vseh poddrevesih in seveda na koncu tudi celemu drevesu.

Primer 6.1. Zanima nas velikost drevesa, torej število elementov, ki jih drevo vsebuje. To dobimo tako, da listom drevesa v predpomnilnik damo vrednost 1, na vseh ostalih vozliščih pa vrednost dobimo tako, da seštejemo vse vrednosti poddreves.



Slika 7. Primer računanja velikosti 2-3 drevesa.

Primer 6.2. Zanima nas velikost maksimalnega elementa v drevesu. V predpomnilnike listov shranimo kar vrednost samih listov, vrednosti v vseh nadaljnjih vozliščih pa izračunamo z operacijo, ki vrne maksimum vrednosti poddreves. Vrednost v korenu bo naša rešitev.



Slika 8. Primer iskanja velikosti maksimalnega elementa 2-3 drevesa.

Za vsako vozlišče in vsako drevo bomo želeli imeti avtomatično izračunane mere. Če imamo vozlišče z dvema sinovoma a in b ter smo na monoidu z asociativno

operacijo $\oplus$, je mera vozlišča $\|a\| \oplus \|b\|$. Podobno je mera vozlišča s tremi sinovi a, b in c enaka $\|a\| \oplus \|b\| \oplus \|c\|$.

Mero konice izračunamo direktno s pomočjo *reducijarl*, ki začetni vrednosti 0 dodaja mere elementov konice.

Podobno kot smo izračunali mero vozlišča, sedaj naredimo še za konično drevo. Čeprav imamo na vedno nižjem nivoju opravka z vedno globljimi drevesi, pa ostane mera na vseh nivojih še vedno istega tipa.

Če sedaj globoko drevo predstavimo s posodobljeno funkcijo *Globoko*, ki ima za prvi argument mero drevesa, ostali trije argumenti pa ostanejo predpona, vmesno drevo m in pripona, potem lahko zapišemo tudi nov pametni konstruktor *globoko*, ki sprejme predpono, vmesno drevo in pripono ter nato sam izračuna mero. Mera je enaka vsoti mer $\|predpona\| \oplus \|m\| \oplus \|pripona\|$.

Mera praznega drevesa je 0, mera enojca x pa kar $\|x\|$.

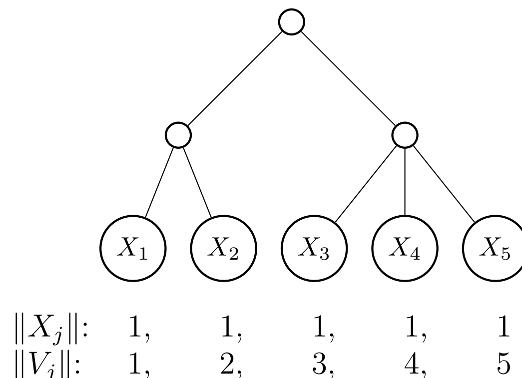
Tudi osnovne operacije, ki ne potrebujejo mer, lahko predelamo v operacije, ki lahko delajo z drevesi, ki imajo predpomnilnike. Novo definicijo zapišemo z novim konstruktorjem, ki vrednosti v predpomnilnikih preprosto ignorira.

Operacija $\oplus$ se na vsakem koraku pokliče samo konstantno mnogokrat, saj so vrednosti v poddrevesih že izračunane, zato shranjevanje podatkov v predpomnilnik ne kvari časovne zahtevnosti in je vseeno, če mere dodamo v definicijo drevesa.

6.1. **Delitev drevesa.** Sedaj, ko imamo predpomnilnike, si lahko ogledamo metodo ravno obratno od združevanja, torej delitev drevesa na dve poddrevesi. Bolj natančno bomo zaenkrat razdelili drevo na tri dele, saj bomo poleg dveh poddreves vrnili tudi element, glede na katerega smo drevo razdelili.

Za začetek bomo definirali metodo *pregled*, na podlagi katere bomo lahko določili, kje razdeliti drevo. Metoda *pregled* preslika zaporedje n elementov $(X_i)_i$ v zaporedje $(V_i)_i$, kjer je $V_j = i \oplus \|X_1\| \oplus \ldots \oplus \|X_j\|$. Pri tem je $\|X_j\|$ vrednost v predpomnilniku, $\oplus$ neka asociativna operacija, i pa neka začetna vrednost.

Primer 6.3. Če bi za *pregled* vzeli podobno funkcijo kot prej, ko smo računali velikost drevesa in za začetno vrednost vzeli 0, bi nam za vsak element vrnila položaj v zaporedju. Vsak element $\|X_j\|$ v zaporedju bi imel namreč mero 1, in ko bi elementom prirejali vrednosti $\|V_j\|$, bi bile te odvisne samo od tega, kateri po vrsti je bil $\|X_j\|$ v zaporedju.



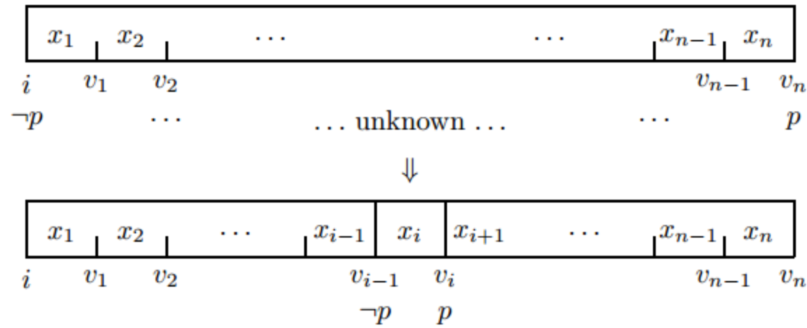
Zdaj bomo lahko uvedli nek predikat. Zanj bo veljalo, da bo imel do nekega elementa vrednost *False*, potem pa se mu bo spremenila na *True*. Kje natanko, bo odvisno od naših zahtev. Tam, kjer se spremeni, razdelimo drevo.

Za našo metodo, ki bo delila drevo, imamo tri osnovne zahteve. Prva je ta, da ima predikat na samem začetku vrednost *False*, če pa pregleda celo drevo, ima vrednost

True. S tem si zagotovimo, da bomo vmes imeli zagotovo vsaj eno točko, kjer se vrednost obrne, torej kjer lahko razdelimo drevo na dva dela.

Druga zahteva je, da če spremenimo novi dobljeni drevesi v seznam in ju združimo skupaj z elementom, da dobimo isto, kot če bi spremenili naše začetno drevo v seznam. To pomeni, da se je vrstni red podatkov res ohranil in da nismo izgubili kakšnih podatkov.

Tretja zahteva pa je, da ima predikat po delitvi na levem drevesu res še vrednost *False*, medtem ko če pregleda še element okoli katerega delimo, ima vrednost *True*.



SLIKA 9. Metoda *razdeliDrevo*. Slika povzeta iz članka [2].

Te zahteve seveda ne preprečujejo predikatu, da bi ta večkrat zamenjal vrednost, je pa najbolj uporabno, če jo zamenja samo enkrat. Enoličnost razcepa si lahko zagotovimo z monotono funkcijo. Mi se bomo ukvarjali z enoličnimi razcepi.

Tako definirana metoda lahko deluje nad poljubnim monoidom. Mi si bomo ogledali samo osnovni problem, to je kako razdeliti drevo glede na i-ti element v drevesu. Torej bo do i-tega elementa imel naš predikat vrednost *False*, ter *True*, ko pogleda še i-ti element.

Skoraj v vsakem primeru bo potrebno v metodi razdeliti konico, zato najprej definiramo pomožno metodo *razdeliKonico*, ki bo prav tako vrnila tri dele.

Metoda *razdeliKonico* najprej pogleda, če je v konici samo en sam enojec. Katere stopnje je ta enojec, je odvisno od tega, kako globoko v drevesu smo. Na prvem nivoju bo stopnje 1, torej bo to kar iskani element. Če ni stopnje 1, bo kasneje definirana metoda *razdeliDrevo* poskrbela, da bomo dobili kot končni rezultat samo naš iskani element. Torej v primeru, da imamo enojec, metoda vrne dva prazna seznama in vmes enojec.

Če je v konici več dreves, pa ločimo dva primera, in sicer če je i-ti element že v prvem drevesu ali pa v katerem od ostalih dreves.

V prvem primeru že prvo drevo vsebuje i-ti element. Takrat metoda *razdeliKonico* vrne najprej prazen seznam, nato to drevo z i-tim elementom, za zadnji del pa preostanek seznama.

V drugem primeru, torej če i-ti element ni v prvem drevesu, najprej izvedemo rekurzivni klic na skrajšanem seznamu brez prvega drevesa, saj vemo, da leži iskani element v enem od dreves v tem seznamu. Ta klic nam vrne tri dele, ki jih označimo z l, x in d. Za novi levi del, ki ga bo vrnila naša *razdeliKonico*, združimo prvo drevo in l. Preostala dva dela ostaneta enaka.

Osnovni problem, ko delimo okoli i-tega elementa, se posploši enostavno s splošnim predikatom. Splošen primer bomo uporabljali tudi kasneje pri uporabi koničnega drevesa.

```
razdeliKonico(predikat, zacVr, s):  ← koda za splošen primer
    a=s[0]
    if length(s)==1: return ([], a, [])
    else:
        as=s[1:]
        zacVr'=zacVr ⊕ ||a||
        if predikat(zacVr'): return ([], a, as)
        else:
            (l, x, d)=razdeliKonico(predikat, zacVr', as)
            return ([a] + l, x, d)
```

Ker pregledujemo konico z leve proti desni, bi se ta razdelila prvič, ko bi se predikat spremenil iz *False* v *True*. Metoda *razdeliDrevo*, ki jo bomo sedaj definirali, deluje podobno, s tem da pri pregledovanju drevesa lahko preskočimo cela poddrevesa, kar pa je nujno, če želimo imeti logaritemsko časovno zahtevnost. Nam pa ne zagotavlja več, da je to res prvi razcep. V našem primeru imamo monoton predikat, torej to ni problem.

Tudi metoda *razdeliDrevo* najprej obravnava najbolj osnoven primer, to je ko za drevo dobimo enojec. Tega ne bomo več delili, zato vrnemo levo in desno dva prazna seznama, vmes pa enojec. Spet je tu enojec lahko 2-3 drevo višje stopnje, ki ga dobimo z rekurzivnim klicem metode globlje po drevesu.

Srce metode pa je seveda razdelitev globokega drevesa. Ločimo tri primere, glede na to v katerem delu drevesa leži i-ti element. Leži lahko v eni izmed konic ali pa v vmesnem drevesu.

Element je v levi konici: Z metodo *razdeliKonico* razdelimo konico na tri dele, ki jih poimenujemo l, x in d. Vemo, da je x naš iskani enojec, torej ga vrnemo kot srednji del. Za levo drevo bomo vrnili l, ki ga je iz seznama potrebno še spremeniti v pravo konično drevo z že poznano funkcijo *vDrevo*. Desno drevo ostane enako originalnemu, le njegova leva konica je sedaj enaka r. Če je r prazen, pametni konstruktor $globoko_L$ drevo popravi, kot smo to naredili pri odstranjevanju elementa.

Element je v vmesnem drevesu: Najprej rekurzivno pokličemo *razdeliDrevo*, ki nam vrne tri dele: ml, mx in md. mx je enojec, v katerem je i-ti element. mx je 2-3 drevo, ki je stopnjo globlje kot element oziroma drevo, ki ga želimo vrniti, saj smo mx dobili z rekurzivnim klicem po stopnjo globljem drevesu. Torej ta enojec spremenimo v seznam dreves, ki so stopnjo plitvejši in na njem pokličemo metodo $razdeliKonico$, ki vrne dele l, x in d, pri čemer je x spet enojec, v katerem je naš element, ampak stopnjo plitvejši. Na vsakem nivoju, kjer rekurzivno pokličemo metodo nivo nižje poskrbimo na koncu, da bomo kot rezultat vseeno dobili drevesa tiste stopnje, s katerimi delamo na tem nivoju. Sedaj imamo torej komponente, ki jih je potrebno združiti v dve drevesi. Prvo drevo dobimo z združitvijo leve konice originalnega drevesa ter delov ml in l, drugo drevo pa iz delov d, mr ter desne konice začetnega drevesa.

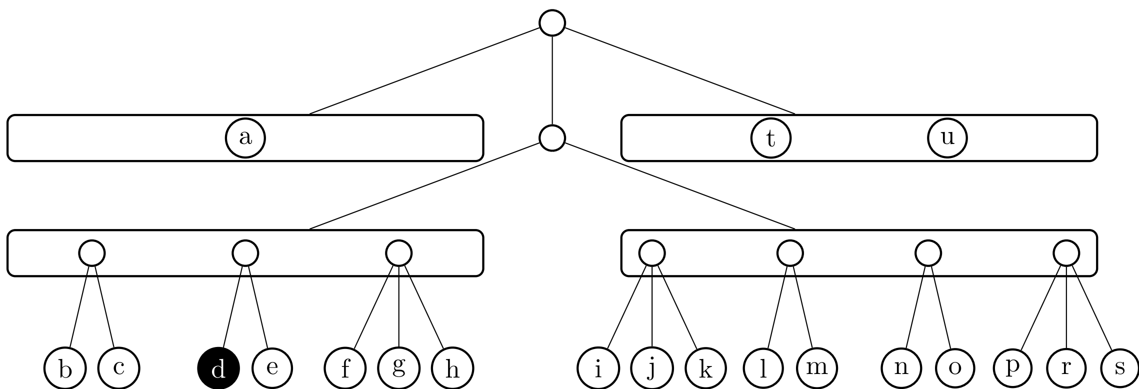
Element je v desni konici: Metoda postopa simetrično, kot če se element nahaja v levi konici.

V vseh primerih se lahko zgodi, da vrnemo enojec višje stopnje, ampak to se vedno zgodi samo, če je bila metoda rekurzivno poklicana. Rekurzivno pa se $razdeliDrevo$

kliče samo, če je element v vmesnem drevesu. V tem primeru pa smo pokazali, da po rekurzivnem klicu metoda poskrbi, da se stopnja drevesa spet zmanjša. Če bi z metodami delali samo na prvem nivoju, bi vsi enojci bili drevesa stopnje 1, torej začetni elementi.

```
razdeliDrevo(predikat, zacVr, dr):  ← koda za splošen primer
    if dr==Enojec x:  return (Prazno, x, Prazno)
    else (dr==(mera, predp, m, prip)):
        vpredp=zacVr ⊕ ‖predp‖
        vm=vpredp ⊕ ‖m‖
        if predikat(vpredp):
            (l, x, d)=razdeliKonico(predikat, zacVr, predp)
            return (vDrevo(l), x, globoko_L(d, m, prip))
        elif predikat(vm):
            (ml, mx, md)=razdeliDrevo(predikat, vpredp, m)
            (l, x, d)=razdeliKonico(predikat, (vpredp ⊕ ‖ml‖), vSeznam(mx))
            return (globoko_D(predp, ml, l), x, globoko_L(d, md, prip))
        else:
            (l, x, d)=razdeliKonico(predikat, vm, prip)
            return (globoko_D(predp, m, l), x, vDrevo(d))
```

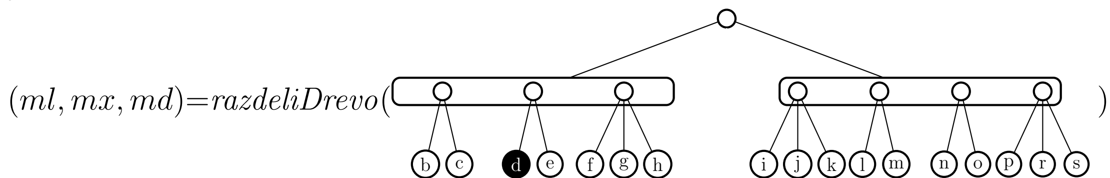
Primer 6.4. Oglejmo si primer delitve drevesa. Podano imamo začetno drevo z označenim iskanim elementom, glede na katerega bomo to drevo razdelili na tri dele.



Slika 10. Začetno drevo.

Prvi korak: Najprej pogledamo, v katerem delu drevesa se nahaja iskani element. Ker je označen element v vmesnem drevesu, najprej poračunamo *razdeliDrevo* na njem.

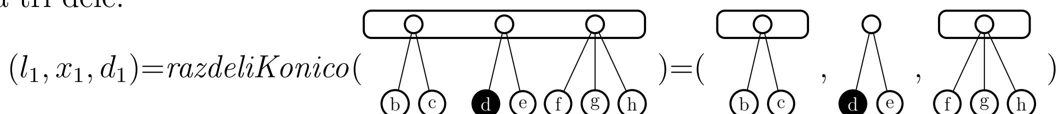
(ml, mx, md)=*razdeliDrevo*(



)

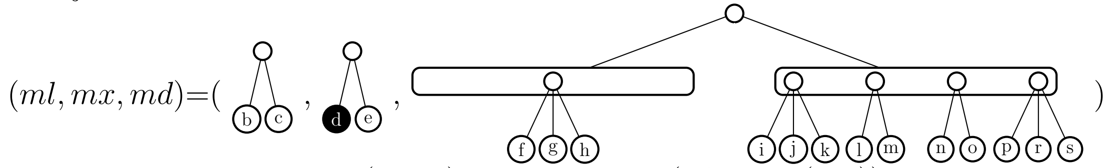
V tem drevesu leži element v predponi, zato jo z metodo *razdeliKonico* razdelimo na tri dele.

(l_1, x_1, d_1)=*razdeliKonico*(



)

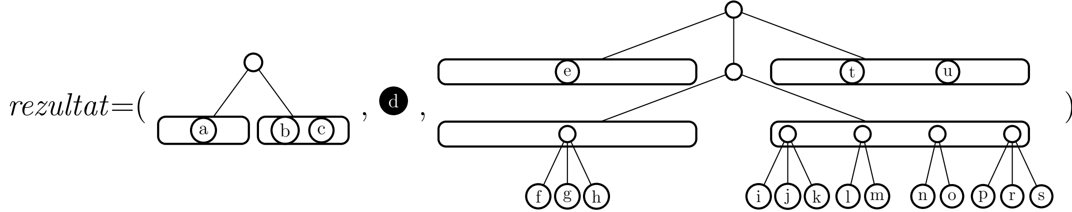
Sedaj lahko do konca izračunamo *razdeliDrevo*.



Drugi korak: Izračunamo (l, x, d)=*razdeliKonico*(*vSeznam*(mx))

(l, x, d)=(, d , e)

Sedaj smo poiskali vse potrebne dele in lahko izračunamo kaj nam vrne funkcija *razdeliDrevo*.



Časovna zahtevnost je sorazmerna z globino elementa, okoli katerega delimo. Povedali smo, da za dostop do njega potrebujemo $O(\log d)$, pri čemer je d razdalja do bližjega konca drevesa, ki ga podamo. Potemtakem je zahtevnost metode *razdeliDrevo* $O(\log(\min\{n_l, n_d\}))$, kjer sta n_l in n_d velikosti dreves, ki ju vrnemo.

Očitno *razdeliDrevo* deli drevo, ki je neprazno. Rekurzivno se funkcija ne more poklicati na praznem drevesu, saj bi se to lahko zgodilo le, če bi bilo drevo m prazno, ampak takrat je $vm = vpredpona$, torej se bo funkcija rekurzivno klicala na predponi in ne na m.

Da je vrednost predikata na začetku drevesa res *False* in na koncu *True*, se preveri samo ob delitvi drevesa na koncih. Če ta dva primera obravnavamo posebej, je torej zadosti zahtevati samo, da je drevo neprazno. Ko drevo razdelimo na tri dele l, x in d, smo prej od predikata zahtevali, da vrne *False*, ko pregleda l. Sedaj pa zahtevamo, da je l prazno drevo ali pa ustreza prejšnji zahtevi. Podobno, ko pregleda predikat še x, mora vrniti *True* ali pa je d prazno drevo. Torej je drevo na vsaki strani lahko prazno in o predikatu ne moremo povedati ničesar, ali pa zadošča zahtevi.

Sedaj lahko definiramo funkcijo *razdeli*, ki bo vrnila samo dve drevesi l in d. Za argumenta sprejme samo predikat in drevo.

Funkcija prazno drevo razdeli na dve prazni drevesi. Če drevo ni prazno in predikat ne vrne *True*, ko pregleda celo drevo, vrne za prvo drevo kar originalno drevo, za drugo pa prazno. Če pa predikat vrne *True*, potem pa razdelimo drevo s funkcijo *razdeliDrevo* (dodamo ji začetno vrednost 0) in dobimo tri dele l, x in d. *razdeli* kot prvo drevo vrne l, za drugo pa d, ki mu še prej z leve dodamo x.

Taka specifikacija je močnejša, saj ne potrebujemo nobenih predpostavk za predikat. V prejšnji (šibkejši) specifikaciji, bi moral predikat vrniti *False* na začetku, čeprav se ta zahteva v kodi nikoli ne pojavi.

Za kasnejšo uporabi si samo še definirajmo funkciji *prvoDrevo*, ki vrne prvo komponento *razdeli* in *drugoDrevo*, ki vrne drugo.

7. Uporaba

S pomočjo koničnega drevesa lahko implementiramo marsikatero strukturo, zato si bomo v zadnjem razdelku ogledali nekaj takih struktur.

7.1. **Seznam z naključnim dostopom.** Ko govorimo o implementaciji seznama z naključnim dostopom, si želimo, da bi naše konično drevo imelo hiter dostop do poljubnega elementa.

Koničnim drevesom dodamo velikosti. Za funkcijo monoida vzamemo navadno seštevanje, torej vsota velikosti m in n je velikost $m + n$. Velikost prazne strukture je 0 (identiteta). Ker potrebujemo informacijo, kje v seznamu se element nahaja, definiramo, da je velikost elementa 1, saj bomo s seštevanjem velikosti elementov po vrsti dobili ravno njihov položaj v seznamu.

S takim monoidom izračunamo vsakemu vozlišču drevesa velikost. Če predstavimo seznam s koničnim drevesom, imamo za vsak element v seznamu zraven tako še podatek, kje se nahaja.

Če nas zanima *dolzina* seznama, pogledamo velikost drevesa, ki ta seznam predstavlja, kar pa je že izračunano, torej imamo ta podatek v konstantni časovni zahtevnosti.

Za dostop do i-tega elementa seznama si pomagamo s funkcijo *razdeliDrevo*. Seznam, ki je predstavljen s koničnim drevesom, razdelimo s funkcijo *razdeliDrevo* na tri dele, levo in desno drevo ter iskani element. Podatkov, kakšna sta levi in desni seznam, ne potrebujemo, zanima nas samo vmesni del, ki je rešitev. Da lahko pokličemo funkcijo, definiramo predikat kot funkcijo, ki vrne *False*, dokler je velikost pregledanega seznama manjša od velikosti i-tega elementa, in *True*, če je večja ali enaka. Predikat je v tem primeru monotona funkcija in nam zagotavlja enoličen razcep seznama.

Če namesto *rezdeliDrevo* vzamemo funkcijo *razdeli*, dobimo delitev seznama na dva dela.

7.2. **Max-prioritetna vrsta.** *Max-prioritetna vrsta* je vrsta, v kateri imajo elementi ob sebi še podatek, ki mu pravimo *prioriteta*. Operacija, ki odstrani element, odstrani tistega z največjo prioriteto.

Funkcija, ki jo bomo uporabljali pri tem primeru, je maksimum. Maksimum je asociativna funkcija, ki ji za identiteto priredimo spremenljivko *mNes* (minus neskončno). Dobili smo monoid *Prio*.

Vsakemu elementu x priredimo vnaprej določeno prioriteto *Prio* x. Prazna struktura ima prioriteto *mNes*. V vsakem vozlišču imamo maksimalno vrednost prioritet poddreves.

Osnovni operaciji na prioritetni vrsti sta dodajanje in odstranjevanje elementa. Dodajanje poteka povsem enako kot osnovno dodajanje brez dodanih prioritet. Pri odstranjevanju elementa pa si zopet pomagamo s funkcijo *razdeliDrevo*. Odstranjujemo element z največjo prioriteto. V primeru, da imamo takih elementov več, odstranimo tistega, na katerega smo naleteli prej. Če poskrbimo, da dva elementa nimata enakih prioritet, dobimo enolično odstranjevanje elementa. Funkcija bo vrnila element, ki ga odstranjujemo, in vrsto brez tega elementa.

Funkciji *razdeliDrevo* moramo podati tri podatke: predikat, začetno vrednost in vrsto, ki jo delimo. Vrsto že imamo, za začetno vrednost pa vzamemo *mNes*. Predikat bo funkcija, ki bo vrnila *False*, če je vrednost vrste oz. poddrevesa, ki jo predstavlja, manjša od velikosti celotnega drevesa, in *True*, če bo večja ali enaka. Vemo, da večja ne more biti, torej bo spremenila vrednost, ko bo naletela na element z največjo prioriteto.

razdeliDrevo vrne dve drevesi l in d ter vmesni element x. Naša funkcija sedaj vrne x ter l in d, ki ju še prej združimo s funkcijo $\bowtie$.

7.3. **Urejen seznam.** Kot že ime pove, je to seznam, katerega elementi so urejeni po nekem kriteriju. Lahko so urejeni po abecedi, koledarsko ali pa imamo povsem poljubno urejanje.

Monoid *Kljuc*, s katerim si bomo pomagali, bo imel funkcijo, ki za rezultat vedno vrne drugi argument, razen če je drugi element ravno identiteta. Takrat vrne prvi argument. Za identiteto proglasimo spremenljivko *nimaKljuca*. Vsak element bo imel ob sebi za podatek nek ključ. Prazna struktura bo imela ključ *brezKljuca*, vse ostale pa neko vrednost.

Ključ vsakega poddrevesa bo torej ključ zadnjega od njegovih poddreves.

Dodajanje elementa urejenemu seznamu moramo definirati na novo, saj moramo dodati element na točno določeno pozicijo, da ohranimo urejenost. Zopet si pomagamo s funkcijo *razdeli*. Predikat bo vrnil *False*, če je ključ manjši od ključa *Kljuc* x, kjer je x element, ki ga dodajamo. Predikat bo vrnil *True*, če je ključ večji ali enak ključu *Kljuc* x. Funkcija *razdeli* vrne dve drevesi, ki ju označimo z l in d. Drevesu d z leve nato dodamo element x. Drevo l zdaj združimo z novo dobljenim drevesom s pomočjo funkcije $\bowtie$. Dobili smo novo drevo z elementom x, ki predstavlja urejen seznam.

S pomočjo funkcije *razdeli* lahko tudi odstranimo elemente s ključem *Kljuc* x. Prvič uporabimo *razdeli* enako kot pri dodajanju. Spet dobimo drevesi l in r. Drevo r nato spet razdelimo na dva dela s pomočjo podobnega predikata, ki zdaj vrne *False*, če je ključ manjši ali enak od ključa *Kljuc* x, in *True*, če je večji. S tem smo dobili dve drevesi l' in r'. Drevo l' vsebuje vse elemente, ki imajo za vrednost *Kljuc* k. Če združimo drevesi l in r' dobimo drevo brez elementov, ki smo jih brisali. To drevo še vedno predstavlja urejen seznam.

Če bi želeli odstraniti samo en element, bi to storili podobno kot smo pri Max-prioritetni vrsti s pomočjo funkcije *razdeliDrevo*.

Tudi sicer je ta monoid *Kljuc* nekakšna optimizacija monoida *Prio*, saj če imamo urejen seznam, se funkcija maksmimuma poenostavi na funkcijo, ki vedno izbere drugi argument, *brezKljuca* pa ima enako vlogo kot *mNeg*. Prav tako imamo takojšen dostop do največjega in najmanjšega elementa, saj le-ta ležita na koncih seznama.

Če se vrnemo nazaj na operacije na urejenem seznamu, si oglejmo še spajanje dveh seznamov. Recimo tej funkciji *spoji*. Pomagamo si s funkcijo $pogled_L$.

Združujemo dva seznama, ki ju označimo z x in y. Najprej izvedemo $pogled_L$ na seznamu y. Če $pogled_L$ vrne, da je seznam prazen, takrat *spoji* vrne seznam x. Če pa $pogled_L$ vrne nek element a in preostanek seznama y', potem izvedemo še *razdeli* na seznamu x s predikatom, ki vrne *False*, če je ključ, ki ga pregleduje, manjši ali enak ključu *Kljuc* a, in *True*, če je večji. Funkcija *razdeli* vrne dve drevesi l in d. Naša funkcija *spoji* se bo sedaj rekurzivno poklicala, kjer namesto na x in y deluje na y' ter d. Ta *spoji* bo vrnil nek urejen seznam, ki mu sedaj dodamo z leve še a. Označimo dobljeni seznam s s. Za konec še s funkcijo $\bowtie$ združimo drevesi l in s.

Tako definirano spajanje razdeli začetna seznama v minimalno število segmentov, ki morajo biti preurejeni, da obdržimo urejenost rezultata.

7.4. **Intervalno drevo.** V tem delu bomo s pomočjo koničnih dreves implementirali *intervalno drevo*. To je urejeno drevo, ki vsebuje intervale, pri čemer se ti intervali lahko med seboj prekrivajo. To je podatkovna struktura, ki nam omogoča učinkovito poiskati vse intervale, ki se prekrivajo z vnaprej danim intervalom ali pa vsebujejo neko dano točko [6]. Delamo z zaprtimi intervali.

Mi bomo s pomočjo koničnih dreves poiskali interval, ki se prekriva z danim intervalom v $O(\log n)$, oziroma v $O(m \log \frac{n}{m})$, če želimo vrniti vse take intervale, pri čemer je n število vseh intervalov v drevesu, m pa število tistih, ki se prekrivajo z danim intervalom [2].

Pri tej aplikaciji bomo za vsak element imeli podatek z dvema komponentama, zato si bomo pomagali s produktom dveh monoidov. Prva komponenta bo začetna točka intervala, drugi podatek pa končna. Intervale bomo imeli v drevesu urejene glede na začetne točke, od najmanjše do največje, zato za prvi monoid vzamemo monoid *Kljuc*, s tem da vsakemu intervalu avtomatično priredimo ključ, ki bo kar začetna točka intervala. Kot drugo komponento podatka želimo imeti končne točke intervalov, zato lahko uporabimo monoid *Prio* in definiramo končne točke intervalov kot prioritete elementov.

Definiramo si nova predikata *vsaj* in *vecji*. Predikat *vsaj*, ki bo sprejel realno število k in podatek, ki ima za drugo komponento n, bo vrnil *False*, če je *Prio* k (prioriteta velikosti k) večja od n in *True*, če je manjša ali enaka. Predikat *vecji* pa bo sprejel število k in podatek, ki ima n za prvo komponento. Ta predikat bo vrnil *False*, če je *Kljuc* k večji ali enak n, in *True*, če je manjši.

Prvi predikat *vsaj* bomo uporabili v funkciji *poisciInterval*, ki bo, kot pove ime, poiskala interval, ki se seka z danim intervalom I. Vhodni podatki bosta s koničnim drevesom predstavljeno intervalno drevo *intDrevo* ter interval I. Funkcija bo vrnila interval iz drevesa, ki se seka z intervalom I, če tak interval obstaja, sicer ne vrne nič.

Funkcija *poisciInterval* najprej preveri, kaj vrne predikat *vsaj*, če je prvi argument začetna točka intervala I, drugi argument pa mera drevesa *intDrevo*, torej (*Kljuc intDrevo*, *Prio intDrevo*). *Kljuc* bo najmanjša začetna točka vseh intervalov, *Prio* pa največja končna. Če *vsaj* vrne *False*, I leži desno od vseh intervalov v drevesu, torej se z nobenim ne prekriva, zato *poisciInterval* ne vrne ničesar in se s tem zaključi iskanje.

Če vrne predikat *True*, pa računamo dalje. S funkcijo *razdeliDrevo* razdelimo *intDrevo* na tri dele. Začetna vrednost bo kar 0, za predikat pa vzamemo *vsaj*, ki ima za prvi argument začetno točko intervala I. Drugi argument je spremenljivka. Od rezultatov, ki jih vrne *razdeliDrevo*, nas zanima samo vmesni interval, ki ga označimo z x. To je prvi interval, na katerega smo naleteli, ki ima končno točko večjo ali enako začetni točki intervala I. Ker je to prvi interval, ima od vseh intervalov, ki ustrezajo temu pogoju, najmanjšo začetno točko. Treba je samo še preveriti, da je začetna točka x manjša ali enaka končni točki I. Če je, je x naša rešitev, sicer rešitve ni in *poisciInterval* ne vrne ničesar. Ker ima x od vseh možnih rešitev najmanjšo začetno točko, vemo, da če ta x ni rešitev, da tudi druge rešitve ne obstajajo, saj imajo le-te kvečjemu večjo začetno točko in zagotovo ne ustrezajo pogoju, da mora biti začetna točka intervala manjša od končne točke I.

Podobno lahko definiramo funkcijo *vsiIntervali*, ki vrne seznam vseh intervalov, ki se prekrivajo z danim intervalom I.

Najprej vzamemo končno točko I-ja in na njej uporabimo funkcijo *vecji*, da dobimo predikat. Poleg tega predikata vzamemo sedaj še naše drevo in na njiju lahko izvedemo funkcijo *prvoDrevo*, ki vrne novo drevo dr_1.

Na tem mestu uvedemo novo funkcijo *najdiInt*, ki dobi za argument drevo dr_n. Podobno bo ta funkcija najprej naredila predikat s pomočjo *vsaj*, ki za argument dobi začetko točko I-ja. Če uporabimo sedaj funkcijo *drugoDrevo* s tem predikatom in drevesom dr_n, dobimo novo drevo dr_2. S $pogled_L$ pogledamo, če je to drevo prazno.

Če je, vrne funkcija *najdiInt* prazen seznam. Sicer $pogled_L$ vrne njegov prvi element a ter drevo brez tega element dr_a. V tem primeru se *najdiInt* rekurzivno pokliče na dr_a. Ta klic vrne nek seznam sez. Seznamu z leve dodamo b in vse skupaj vrnemo kot rezultat.

Če se vrnemo nazaj na funkcijo *vsiIntervali*. Ta sedaj samo pokliče funkcijo $najdiInt$ na drevesu d_1 in kot rešitev dobimo seznam vseh intervalov, ki se prekrivajo z danim intervalom I.

Funkcija *prvoDrevo* izbere take intervale, ki imajo začetno točko manjšo od končne točke intervala I, s tem da vzame tudi intervale, ki imajo končno točko manjšo od začetne točke I. Teh intervalov smo se znebili potem s pomočjo *drugoDrevo*.

Literatura

[1] H. Daumé, *Yet another Haskell tutorial*, [ogled 25. 8. 2012], dostopno na `http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf`.

[2] R. Hinze in R. Paterson, *Finger trees : a simple general-purpose data structure*, J. Funct. Programming **16** (2006) 197–217.

[3] T. Košir, *Algebra 1*, verzija 4. 2. 2009, [ogled 20. 8. 2012], dostopno na `http://www.fmf.uni-lj.si/~kosir/poucevanje/0910/alg1-fm.html`.

[4] J. Kozak, *Podatkovne strukture in algoritmi*, Matematika – fizika **27**, DMFA Slovenije, Ljubljana, 1997.

[5] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, Cambridge, 1999.

[6] *Interval tree*, [ogled 24. 8. 2012], dostopno na `http://en.wikipedia.org/wiki/Interval_tree`.