UNIVERZA V MARIBORU

FAKULTETA ZA NARAVOSLOVJE IN MATEMATIKO

Oddelek za matematiko in računalništvo

MAGISTRSKO DELO

Nina Turnšek

Maribor, 2022

UNIVERZA V MARIBORU

FAKULTETA ZA NARAVOSLOVJE IN MATEMATIKO

Oddelek za matematiko in računalništvo

Magistrsko delo

LOMLJENA DREVESA

na študijskem programu 2. stopnje Matematika

Mentor:

doc. dr. Andrej Taranenko

Kandidatka:

Nina Turnšek

Maribor, 2022

ZAHVALA

"Find something that makes you happy, make it the center of your life and then everything else will naturally fit in around it." - Daniel Sloss

Zahvaljujem se mentorju doc. dr. Andreju Taranenku za vso pomoč in usmeritve pri izdelavi magistrske naloge.

Posebna zahvala gre družini in prijateljem, ki so mi vedno stali ob strani, me spodbujali in verjeli vame.

Vsem iskreno hvala.

Lomljena drevesa

program magistrskega dela

V magistrskem delu bo predstavljena podatkovna struktura imenovana lomljena drevesa. Predstavljeni bodo osnovni pojmi za analizo časovne zahtevnosti ter teoretične (amortizirane) zahtevnosti osnovnih operacij lomljenih dreves in primerjava z drugimi uravnoteženimi dvojiškimi iskalnimi drevesi. Implementiran bo lasten razred za lomljena drevesa ter analizirana bo hitrost delovanja operacij.

Osnovni viri:

1. Daniel D. Sleator and Robert E. Tarjan. *Self-adjusting binary search trees.* Journal of the ACM, 32(3):652-686, 1985.

2. Robert L. Kruse and Alexander J. Ryba. *Data Structures and Program Design in C++.* Prentice Hall, 2000.

3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms, Third Edition.* MIT Press, 2009.

doc. dr. Andrej Taranenko

TURNŠEK, N.: Lomljena drevesa.
Magistrsko delo, Univerza v Mariboru, Fakulteta za naravoslovje in matematiko, Oddelek za matematiko in računalništvo, 2022.

IZVLEČEK

V magistrskem delu je predstavljena podatkovna struktura imenovana lomljeno drevo. Gre za dvojiško iskalno drevo, kjer se oblika drevesa spremeni po vsakem posegu (operaciji) v drevo. Vozlišče, nad katerim izvajamo poljubno operacijo, je na koncu operacije vedno v korenu drevesa. Postopku, ki vozlišče premakne v koren drevesa, pravimo *lomljenje*. Namen lomljenih dreves je, da so podatki, ki jih pogosto uporabljamo, hitro dostopni. Tako podatki, ki jih večkrat uporabljamo, ostanejo bližje vrha drevesa in jih ob naslednji uporabi hitreje najdemo. Podatki, ki so redko v uporabi, se nahajajo nižje v drevesu.

Na podlagi amortizirane časovne zahtevnosti je analizirana hitrost delovanja osnovnih operacij lomljenih dreves. Amortizirana časovna zahtevnost je povprečen čas posamezne operacije v najslabšem zaporedju operacij. V magistrskem delu je predstavljen tudi implementiran program za lomljena drevesa, v katerem so definirane osnovne operacije na lomljenih drevesih. Nazadnje je narejena še analiza hitrosti delovanja operacij implementiranega programa za lomljena drevesa in primerjava lomljenih dreves z drugimi uravnoteženimi drevesi.

Ključne besede: lomljeno drevo,
lomljenje,
amortizirana časovna zahtevnost,
uravnotežena drevesa.

Math. Subj. Class. (2010): 68P05, 05C05, 68W01, 68W40.

TURNŠEK, N.: Splay trees.
Master Thesis, University of Maribor, Faculty of Natural Sciences and Mathematics, Department of Mathematics and Computer Science, 2022.

ABSTRACT

The master's thesis presents the data structure called splay tree. It is a binary search tree, where the shape of the tree changes after each intervention (operation) in the tree. At the end of each operation, the node over which the operation is performed is always in the root of the tree. The process of moving a node to the root of a tree is called *splaying*. The purpose of splay trees is to make frequently used data quickly accessible. Thus, the data we use multiple times remains near the top of the tree and is then found faster. Data that is rarely used is located lower in the tree.

Using the amortized time complexity, we analyse the speed of basic operations on splay trees. Amortized time complexity is the average time of an individual operation in the worst sequence of operations. In the master's thesis an implementation for splay trees is also presented. Finally, the time complexity analysis is made for the operations of the implementation and a comparison of splay trees with other balanced trees is given.

Keywords: splay tree,
splaying,
amortized time complexity,
balanced trees.

Math. Subj. Class. (2010): 68P05, 05C05, 68W01, 68W40.

Kazalo

1 Osnovna terminologija **1**

1.1 Drevesa 1

1.2 Dvojiško drevo 2

1.2.1 Statična predstavitev dvojiških dreves 4

1.2.2 Dinamična predstavitev dvojiških dreves 5

1.2.3 Pregled dvojiških dreves 6

1.3 Dvojiška iskalna drevesa 7

1.3.1 Osnovne operacije nad dvojiškim iskalnim drevesom 8

2 Lomljena drevesa **12**

2.1 Lomljenje 12

2.1.1 Lomljenje od spodaj navzgor 14

2.1.2 Lomljenje od zgoraj navzdol 20

2.2 Operacije na lomljenih drevesih 30

2.2.1 Iskanje vozlišča z danim ključem 30

2.2.2 Vstavi vozlišče z danim ključem 32

2.2.3 Odstrani vozlišče z danim ključem 34

3 Amortizirana analiza zahtevnosti **37**

3.1 Agregatna metoda 37

3.2 Računovodska metoda 39

3.3 Metoda potencialov 40

4 Amortizirana zahtevnost lomljenih dreves **43**

4.1 Teoretična amortizirana zahtevnost lomljenih dreves 47

5 Implementacija **53**

5.1 Vhodni podatki 54

5.2 Delovanje programa 55

5.2.1 Metoda *preberiDrevo* 55

5.2.2 Operacija lomljenje od spodaj navzgor 56

5.2.3 Operacija lomljenje od zgoraj navzdol 58

5.2.4 Operacija vstavljanja ključa v drevo 61

5.2.5 Operacija odstranjevanja ključa iz drevesa 63

5.3 Metode z naključno izbiro vozlišč 66

5.3.1 Vstavljanje 66

5.3.2 Odstranjevanje 66

5.3.3 Lomljenje 67

5.4 Analiza delovanja operacij 68

6 Uravnotežena drevesa **74**

6.1 *AVL* drevesa 74

6.1.1 Primerjava lomljenih drevesa z *AVL* drevesi 75

6.2 Rdeče-črna drevesa 76

6.2.1 Primerjava lomljenih drevesa z rdeče-črnimi drevesi 77

6.3 *B*-drevesa 78

6.3.1 Primerjava lomljenih dreves z *B*-drevesi 79

Literatura **81**

Poglavje 1

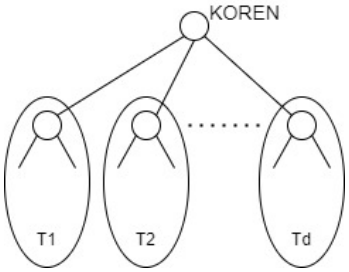
Osnovna terminologija

1.1 Drevesa

V teoriji grafov je drevo definirano kot povezan graf brez ciklov. V magistrskem delu drevo obravnavamo z vidika podatkovnih struktur, kot drevo s korenom.

Definicija 1.1 ([3]) *Drevo je bodisi prazno, bodisi ga sestavlja končna neprazna množica vozlišč, za katero velja, da je eno vozlišče posebej izbrano. To vozlišče imenujemo koren. Če koren odstranimo, drevo razpade na disjunktno unijo dreves, ki jim pravimo poddrevesa.*

Drevo predstavimo grafično, kot prikazuje slika 1.1.

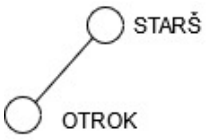


Slika 1.1: Grafična predstavitev podatkovne strukture drevo.

Drevesa lahko uporabljamo za prikaz različnih vsakdanjih podatkov, kot so družinsko drevo, drevo živalskih vrst, hierarhija vojske ali podjetja, itn.

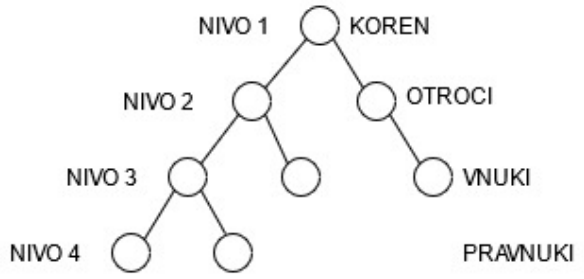
Število poddreves v danem vozlišču imenujemo stopnja vozlišča. Vozlišče katerega stopnja je 0, imenujemo list.

V drevesu so vozlišča med sabo v relaciji starš – otrok. Npr. korensko vozlišče je starš, vozlišče, ki je povezano s korenom je otrok (slika 1.2).



Slika 1.2: Prikaz relacije starš – otrok v drevesu.

Vozlišča v drevesu so razporejena po nivojih glede na oddaljenost od korena. Na sliki 1.3 vidimo, da je koren na nivoju 1, njegovi otroci na nivoju 2, vnuki na nivoju 3, itn. Globina drevesa je največji nivo drevesa.



Slika 1.3: Prikaz nivojev drevesa in relacij med vozlišči.

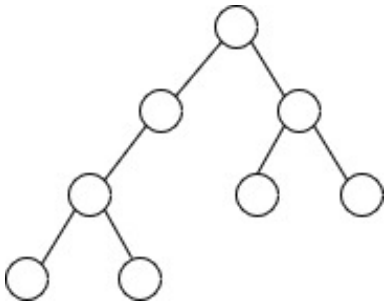
Drevo, v katerem so otroci vsakega vozlišča urejeni, imenujemo urejeno drevo. To pomeni, da v urejenem drevesu ločimo prvega otroka, drugega otroka, itn. [3].

1.2 Dvojiško drevo

Dvojiško drevo je urejeno drevo, v katerem ima vsako vozlišče največ dva otroka in je zelo pogosta in uporabna oblika drevesa. Primer dvojiškega drevesa je prikazan na sliki 1.4.

Definicija 1.2 ([3]) *Dvojiško drevo je bodisi prazno, bodisi ga sestavlja končna množica vozlišč, od katerih je eno vozlišče koren, druga vozlišča pa razpadejo v dve disjunktni množici, levo poddrevo in desno poddrevo, ki sta prav tako dvojiški drevesi.*

Koren levega poddrevesa vozlišča v je levi otrok vozlišča v, koren desnega poddrevesa vozlišča v je desni otrok vozlišča v.



Slika 1.4: Primer dvojiškega drevesa.

V nadaljevanju je predstavljenih nekaj uporabnih lastnosti dvojiških dreves.

Trditev 1.3 ([3]) *V dvojiškem drevesu je na nivoju i največ 2^{i-1} vozlišč.*

Dokaz. Indukcija po i:

1. $i = 1$: na nivoju 1 je samo koren in tudi $2^{1-1} = 2^0 = 1$.
2. Predpostavimo, da je na nivoju i največ 2^{i-1} vozlišč. Ker imajo vsa vozlišča največ dva otroka, za nivo $i+1$ dobimo največ $2 \cdot 2^{i-1} = 2^i$ vozlišč.

□

Trditev 1.4 ([3]) *Naj bo h globina in n število vozlišč dvojiškega drevesa D. Potem v dvojiškem drevesu D velja:*

$$n \leq 2^h - 1.$$

Dokaz. Po trditvi 1.3 je na nivoju i največ 2^{i-1} vozlišč. Zato dobimo

$$n \leq \sum_{i=1}^{h} 2^{i-1} = \frac{2^h - 1}{2 - 1} = 2^h - 1.$$

□

Izrek 1.5 ([3]) *Naj bo h globina in n število vozlišč dvojiškega drevesa D. Potem v dvojiškem drevesu D velja*

$$\lceil \log_2(n+1) \rceil \leq h \leq n.$$

Dokaz. Iz trditve 1.4 dobimo

$$n \leq 2^h - 1$$

$$2^h \geq n + 1$$

$$h \geq \lceil \log_2(n+1) \rceil.$$

Za zgornjo mejo skonstruiramo dvojiško drevo, ki ima ob danem številu vozlišč največjo možno globino. To dvojiško drevo je izrojeno drevo, kjer ima vsako vozlišče natanko enega otroka. □

Drevo je levo poravnano, če se nivoji listov razlikujejo za največ ena in se ta sprememba zgodi največ enkrat, ko potujemo po listih od leve proti desni.

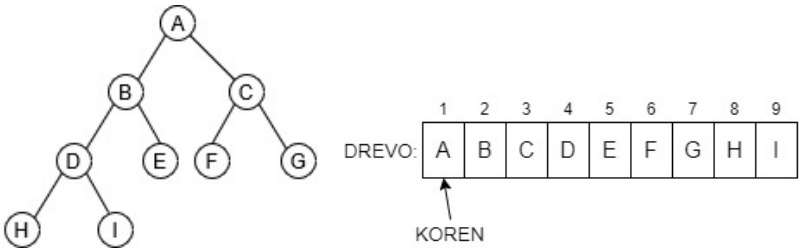
Pravimo, da je nivo i dvojiškega drevesa poln, če je na njem 2^{i-1} vozlišč. Če ima dvojiško drevo vse nivoje polne, temu drevesu pravimo polno dvojiško drevo [3].

Posledica 1.6 ([3]) *Naj bo h globina in n število vozlišč dvojiškega drevesa D. V polnem dvojiškem drevesu D velja*

$$n = 2^h - 1 \text{ in } h = \log_2(n+1).$$

1.2.1 Statična predstavitev dvojiških dreves

Osnovna ideja za statično predstavitev dvojiških dreves je uporaba polja, pri tem predpostavimo, da so elementi indeksirani od 1 naprej. Vozlišča dvojiškega drevesa razporedimo v elemente polja po nivojih od leve proti desni. Koren je na nivoju 1, zato ga vstavimo v polje na indeks 1. Njegova otroka sta na nivoju 2, zato levega otroka vstavimo v polje na indeks 2 in desnega otroka na indeks 3. Vnuki so na nivoju 3, zato jih (gledano od leve proti desni) vstavimo v polje na indekse od 4 do 7, itn. Na sliki 1.5 je prikazan primer statične predstavitve dvojiškega drevesa s poljem.



(a) (b)

Slika 1.5: (a) Primer dvojiškega drevesa. (b) Statična predstavitev dvojiškega drevesa iz (a).

V splošnem za vozlišče na indeksu i velja, da je njegov starš na indeksu $stars_i$, kjer je:

$$stars_i = \begin{cases} \lfloor \frac{i}{2} \rfloor, & \text{vozlišče na indeksu } i \text{ ni koren,} \\ -1, & \text{vozlišče na indeksu } i \text{ je koren.} \end{cases}$$

Vozlišče na indeksu i ima levega otroka na indeksu $levi\_otrok_i$, kjer je:

$$levi\_otrok_i = \begin{cases} 2i, & \text{vozlišče na indeksu } i \text{ ima levega otroka,} \\ -1, & \text{vozlišče na indeksu } i \text{ nima levega otroka.} \end{cases}$$

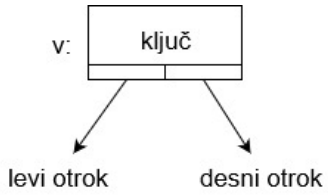
Vozlišče na indeksu i ima desnega otroka na indeksu $desni\_otrok_i$, kjer je:

$$desni\_otrok_i = \begin{cases} 2i+1, & \text{vozlišče na indeksu } i \text{ ima desnega otroka,} \\ -1, & \text{vozlišče na indeksu } i \text{ nima desnega otroka.} \end{cases}$$

Statična predstavitev drevesa je primerna za gosta drevesa z n vozlišči, torej za polna drevesa ali levo poravnana drevesa, saj za njihovo predstavitev potrebujemo polje velikosti n, kar predstavlja linearno zahtevnost. Če statično predstavitev uporabimo na izrojenem drevesu z n vozlišči, potrebujemo polje velikosti $2^{n-1} = O(2^n)$, kar pa predstavlja eksponentno zahtevnost [3].

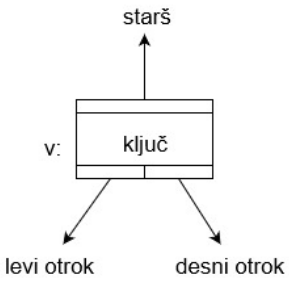
1.2.2 Dinamična predstavitev dvojiških dreves

Primernejša predstavitev dvojiških dreves je dinamična predstavitev, kjer uporabimo kazalce. Vozlišče je tako sestavljeno iz podatka (ključa) ter dveh kazalcev, kjer eden kaže na levega, drugi pa na desnega otroka. Pri dinamični predstavitvi drevesa je posamezno vozlišče predstavljeno, kot je vidno na sliki 1.6.



Slika 1.6: Oblika vozlišča v, če drevo predstavimo dinamično.

Za predstavitev dvojno povezanega drevesa vozlišču dodamo še kazalec, ki kaže na njegovega starša [3]. Tako ima vozlišče obliko, kot je prikazana na sliki 1.7.



Slika 1.7: Oblika vozlišča v v dvojno povezanem drevesu.

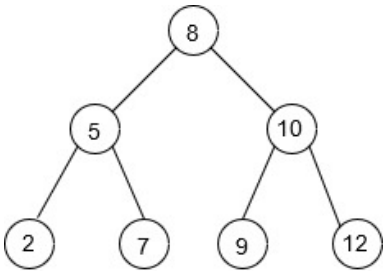
1.2.3 Pregled dvojiških dreves

Z besedo *obisk* označimo poljubno operacijo, ki jo izvedemo v posameznem vozlišču. Ta operacija je lahko npr. izpis vrednosti, izračun, ki temeljni na vrednosti v vozlišču, ali katera koli druga operacija. Če obiščemo vsa vozlišča drevesa v nekem vrstnem redu, pravimo, da smo drevo pregledali.

Predpostavimo, da pri pregledu damo prednost levemu otroku pred desnim in pregled vedno začnemo v korenu drevesa. Ločimo tri vrste pregledov, ki jih definiramo rekurzivno [3]:

- starš prej (SLD): najprej obiščemo vozlišče v, potem njegovo levo poddrevo in nato še njegovo desno poddrevo,
- starš vmes (LSD): najprej obiščemo levo poddrevo vozlišča v, potem vozlišče v in na koncu še njegovo desno poddrevo,
- starš potem (LDS): najprej obiščemo levo poddrevo vozlišča v, potem njegovo desno poddrevo in na koncu še vozlišče v.

Zgled. Na sliki 1.8 je primer dvojiškega drevesa. Operacija obisk naj izpiše vrednost posameznega vozlišča, pri tem izvedemo preglede SLD, LSD in LDS.



Slika 1.8: Primer dvojiškega drevesa nad katerim izvedemo pregled drevesa.

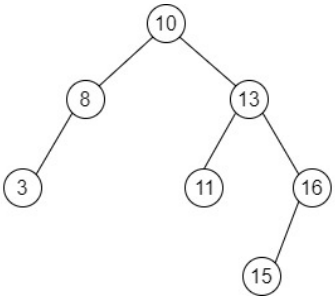
Če nad tem drevesom izvedemo vse tri vrste pregledov, dobimo naslednje:

- SLD: 8, 5, 2, 7, 10, 9, 12,
- LSD: 2, 5, 7, 8, 9, 10, 12,
- LDS: 2, 7, 5, 9, 12, 10, 8.

1.3 Dvojiška iskalna drevesa

Definicija 1.7 ([5]) *Dvojiško iskalno drevo (DID) je dvojiško drevo, v katerem za vsako vozlišče x velja, da so v levem poddrevesu od x ključi, ki so manjši od ključa v x in v desnem poddrevesu od x so ključi, ki so večji od ključa v x. Prav tako sta levo in desno poddrevo od x tudi dvojiški iskalni drevesi.*

Na sliki 1.9 je primer dvojiškega iskalnega drevesa.

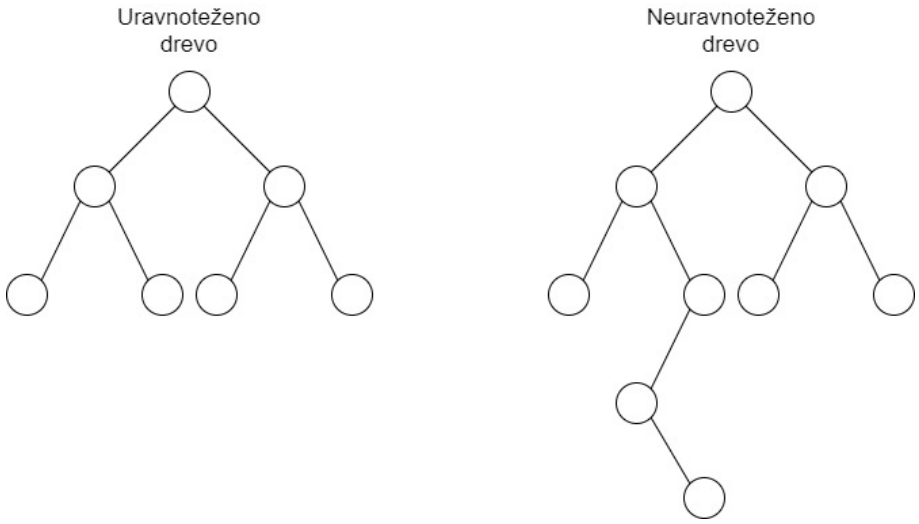


Slika 1.9: Primer dvojiškega iskalnega drevesa.

Dvojiška iskalna drevesa običajno predstavimo dinamično (glej 1.2.2).

Dvojiško iskalno drevo je uravnoteženo, če za vsako vozlišče v velja, da se globini levega in desnega poddrevesa od v razlikujeta za največ 1. Sicer pravimo, da je drevo neuravnoteženo.

Na sliki 1.10 je na levi strani primer uravnoteženega drevesa in na desni strani primer neuravnoteženega drevesa.



Slika 1.10: Primer uravnoteženega (levo) in neuravnoteženega (desno) drevesa.

1.3.1 Osnovne operacije nad dvojiškim iskalnim drevesom

Nad dvojiškimi iskalnimi drevesi lahko izvedemo različne operacije. Med najpogostejšimi so iskanje, vstavljanje, odstranjevanje vozlišča z danim ključem.

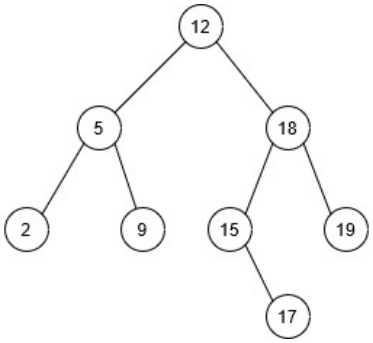
Iskanje vozlišča z danim ključem

Operacija v dvojiškem iskalnem drevesu poišče vozlišče z danim ključem. Parametra algoritma sta dani ključ, ki ga iščemo in kazalec na koren drevesa. Ideja algoritma je, da začnemo v korenu drevesa in se spuščamo levo oz. desno po drevesu v globino. V vsakem vozlišču preverimo, če se ključ vozlišča ujema z iskano vrednostjo. Če je iskani ključ manjši od ključa vozlišča, potem pot nadaljujemo v levem poddrevesu. Če je iskani ključ večji od ključa vozlišča, potem pot nadaljujemo v desnem poddrevesu. Ko najdemo ključ, vrnemo kazalec na ustrezno vozlišče. Če vozlišča s tem ključem v drevesu ni, potem vrnemo kazalec na ničelno vozlišče. Časovna zahtevnost operacije je $O(h)$, kjer je h globina drevesa [1, 3, 5].

Vstavljanje v dvojiško iskalno drevo

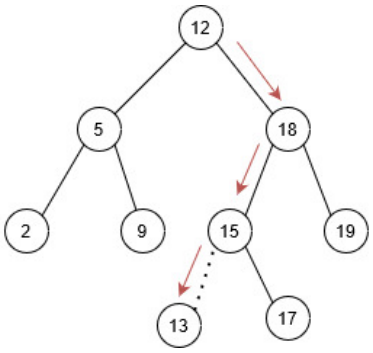
Operacija v dvojiško iskalno drevo vstavi novo vozlišče. Parametra algoritma sta koren drevesa in ključ, ki ga želimo vstaviti v drevo. Novo vozlišče z danim ključem vedno postane list drevesa. Prostor zanj poiščemo tako, da začnemo iskanje v korenu in se spuščamo levo oz. desno po drevesu, dokler ne pridemo do lista. Če je ključ, ki ga želimo vstaviti v drevo manjši od ključa v listu, potem novo vozlišče postane levi otrok lista, sicer postane desni otrok lista. Časovna zahtevnost operacije je $O(h)$, kjer je h globina drevesa [1, 3, 5].

Zgled. V dvojiško iskalno drevo na sliki 1.11 želimo vstaviti vozlišče s ključem 13.



Slika 1.11: Dvojiško iskalno drevo v katerega želimo vstaviti vozlišče s ključem 13.

Novo vozlišče postane levi otrok vozlišča s ključem 15.



Slika 1.12: Dobljeno dvojiško iskalo drevo.

Odstranjevanje vozlišča iz dvojiškega iskalnega drevesa

Pri odstranjevanju vozlišča v iz dvojiškega iskalnega drevesa naletimo na tri možnosti, kako to lahko storimo:

1. če v nima otrok, potem kazalec iz starševskega vozlišča, ki je kazal na vozlišče v, spremenimo, da kaže na ničelno vozlišče,

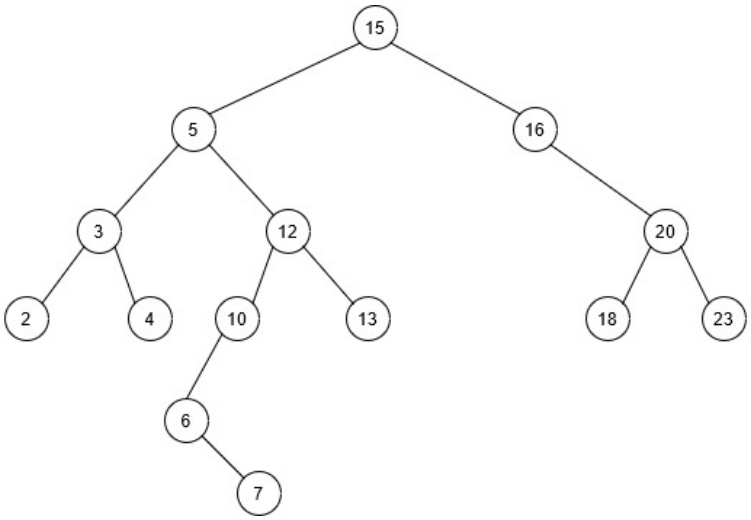
2. če ima v enega otroka, potem kazalec iz starševskega vozlišča, ki je kazal na vozlišče v, preusmerimo, da kaže na edinega otroka vozlišča v,

3. če ima v oba otroka, potem v desnem poddrevesu vozlišča v poiščemo vozlišče x z najmanjšim ključem (t.j. najbolj levo vozlišče desnega poddrevesa od v) in ključ vozlišča v spremenimo v ključ vozlišča x. Nato prvotno vozlišče x odstranimo iz drevesa. Na ta način lažje odstranimo vozlišče x iz drevesa, saj je vozlišče x bodisi

list in nima nobenega otroka (uporabimo točko 1), bodisi ima samo desnega otrok (uporabimo točko 2).

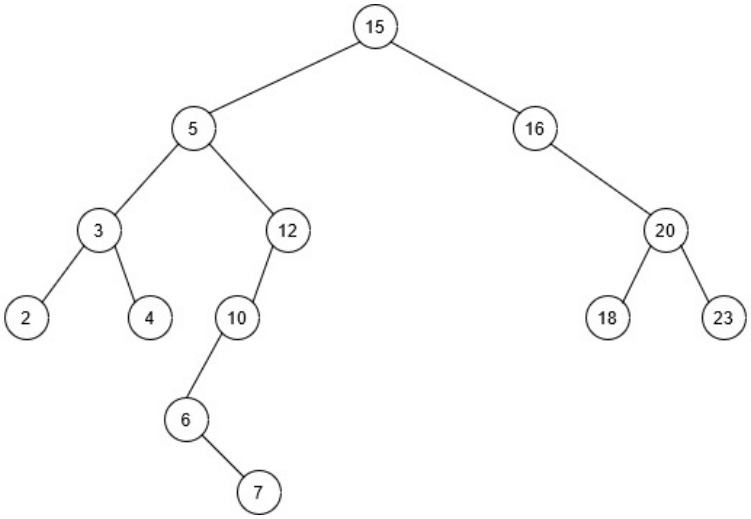
Algoritem za odstranitev vozlišča iz dvojiškega iskalnega drevesa kot parameter prejme koren drevesa in ključ, ki ga želimo odstraniti. Časovna zahtevnost take operacije je $O(h)$, kjer je h globina drevesa [1, 3, 5].

Zgled. Iz dvojiškega iskalnega drevesa 1.13 želimo odstraniti vozlišča s ključi 13, 16 in 5.



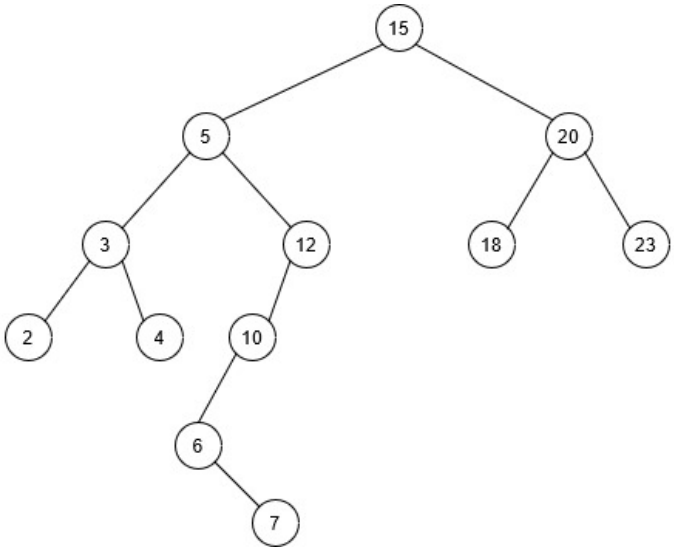
Slika 1.13: Prvotno dvojiško iskalno drevo.

Odstranimo vozlišče 13: kazalec iz starševskega vozlišča, ki je kazal na 13, nastavimo na *nullptr* (ničelno vozlišče).



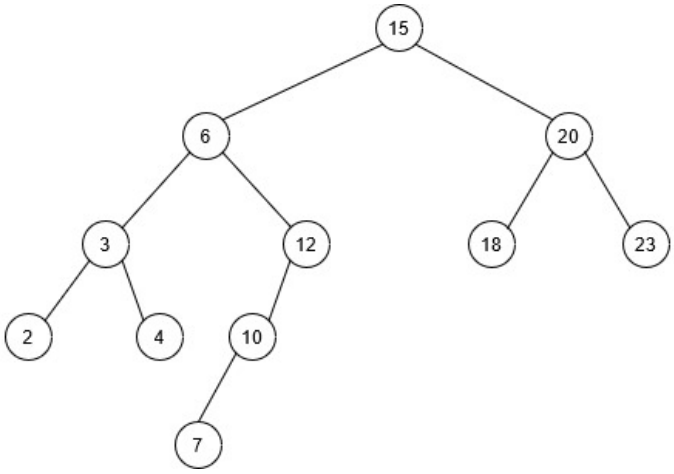
Slika 1.14: Dvojiško iskalno drevo po odstranitvi vozlišča s ključem 13.

Odstranimo vozlišče 16: kazalec iz starševskega vozlišča, ki je kazal na 16, preusmerimo na edinega otroka od 16, torej na vozlišče s ključem 20.



Slika 1.15: Dvojiško iskalno drevo po odstranitvi vozlišča s ključem 16.

Odstranimo vozlišče 5: vozlišče 5 nadomestimo z vozliščem 6, ki je najmanjše vozlišče desnega poddrevesa vozlišča 5 in nato odstranimo prvotno vozlišče 6, ki je list drevesa, tako, da kazalec, ki je kazal na 6, nastavimo na *nullptr* (ničelno vozlišče).



Slika 1.16: Dvojiško iskalno drevo po odstranitvi vozlišča s ključem 5.

Poglavje 2

Lomljena drevesa

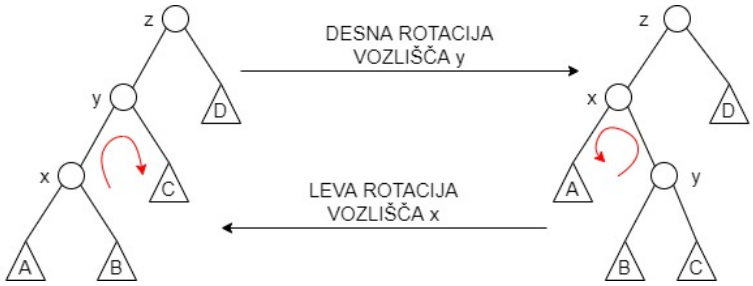
Lomljena drevesa so samoprilagodljiva (angl. self-adjusting) oblika dvojiškega iskalnega drevesa, v katerih po vsaki operaciji vstavljanja ali iskanja vozlišča, to vozlišče premaknemo v koren drevesa. Pri odstranjevanju vozlišča iz drevesa pa v koren premaknemo starša odstranjenega vozlišča. Ostala vozlišča se ustrezno prilagodijo, da naredijo prostor za novi koren. Vozlišča, ki so pogosto uporabljena, so večkrat premaknjena v koren in zato ostanejo blizu korena. Po drugi strani pa so neaktivna vozlišča bolj na dnu drevesa.

Lomljena drevesa lahko postanejo zelo neuravnotežena, tako lahko ena sama operacija iskanja zahteva veliko časa. V nadaljevanju bomo dokazali, da v daljšem zaporedju operacij iskanja lomljena drevesa niso časovno zahtevna. Pri tem bomo uporabili amortizirano analizo časovne zahtevnosti. Na lomljenih drevesih izvajamo operacijo rotacije podobne oblike, kot pri AVL drevesih, vendar s številnimi rotacijami, opravljenimi za vsako vstavitev ali iskanje v drevesu. Rotacije se pravzaprav izvajajo po celotni poti od korena do vozlišča, ki ga iščemo [5, 9].

2.1 Lomljenje

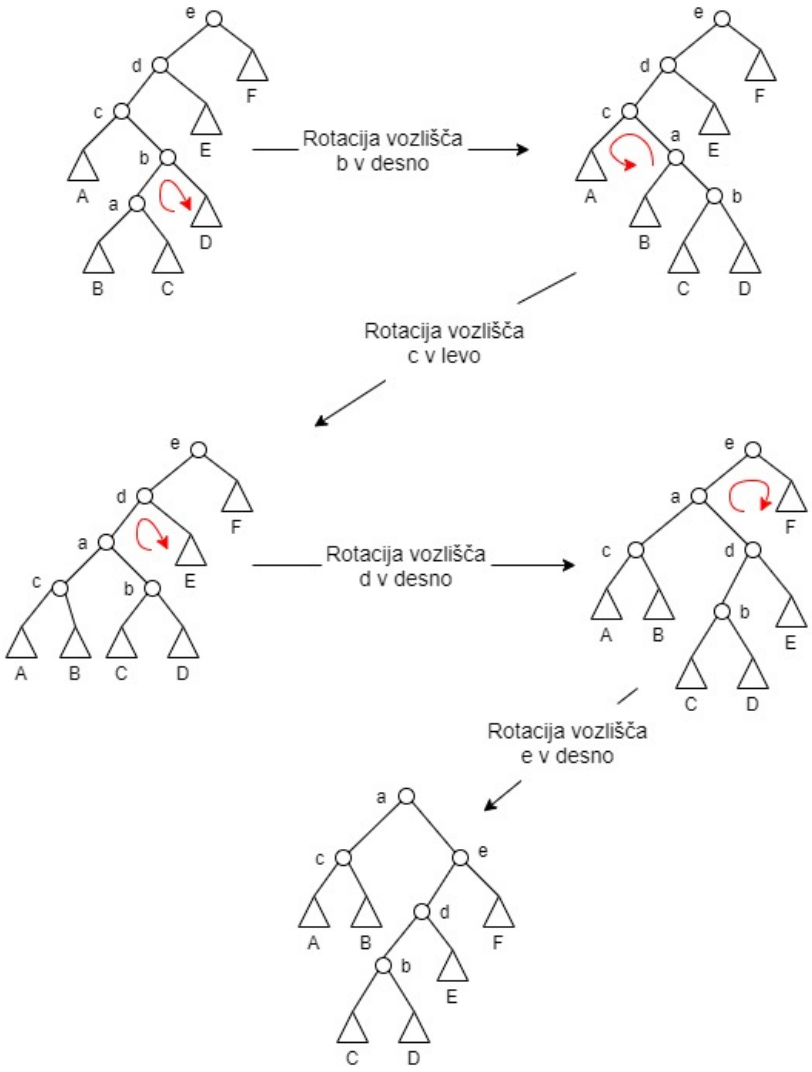
Na lomljenih drevesih po vsaki operaciji premaknemo ustrezno vozlišče v koren drevesa, pri čemer si pomagamo z rotacijami, ki jih izvedemo nad vozliščem x. Uporabimo lahko naslednji dve hevristiki [8, 9]:

- enojna rotacija (angl. single rotation): vozlišče x, ki je starš vozlišča y, zavrtimo v levo. Pri tem se vozlišče x premakne nivo nižje, vozlišče y pa nivo višje. Levo poddrevo od y postane desno poddrevo od x. Opisani rotaciji pravimo leva rotacija. Desna rotacija je definirana podobno. Slika 2.1 prikazuje levo rotacijo vozlišča x in desno rotacijo vozlišča y.



Slika 2.1: Slika prikazuje levo rotacijo vozlišča x in desno rotacijo vozlišča y.

- premik v koren (angl. move to root): glede na položaj vozlišča x v drevesu, izvajamo leve in desne rotacije tako dolgo, dokler x ni v korenu. Na sliki 2.2 je prikazan premik vozlišča a v koren drevesa.



Slika 2.2: Premik vozlišča a v koren drevesa po korakih.

Ko v dvojiškem drevesu izvedemo enojno rotacijo, se nekatera vozlišča premaknejo višje in druga nižje po drevesu. Pri enojni rotaciji vozlišča se to vozlišče premakne nivo nižje, eden izmed njegovih otrok pa se premakne nivo višje. Dvojna rotacija je sestavljena iz dveh enojnih rotacij in se tako lahko eno vozlišče premakne za dva nivoja višje, medtem ko se ostala vozlišča premaknejo za en nivo višje ali nižje. Operacija premik v koren je zaporedje enojnih rotacij (levih in desnih), ki jih izvajamo tako dolgo, dokler izbrano vozlišče ni koren drevesa. Enojno rotacijo vedno izvedemo nad staršem vozlišča, ki ga želimo imeti v korenu drevesa. Med izvajanjem operacije se ostala vozlišča ustrezno premaknejo nižje ali višje v drevesu.

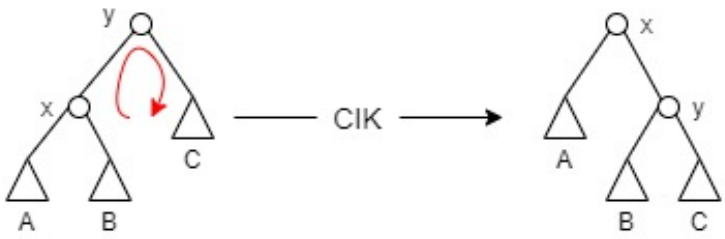
V amoritiziranem smislu (t.j. povprečni čas operacije v najslabšem zaporedju operacij) nobena od teh hevristik ni učinkovita, saj za vsako obstaja poljubno dolgo zaporedje operacij iskanja, tako da je časovna zahtevnost $O(n)$ (več o amortizirani časovni zahtevnosti glej poglavje 3). Namesto tega je zelo učinkovita operacija lomljenje, kjer na vsakem koraku izbrano vozlišče premaknemo dva nivoja višje. Lomljenje lahko izvajamo na dva načina in sicer od spodaj navzgor (glej razdelek 2.1.1), kjer najprej poiščemo ustrezno vozlišče in ga nato premaknemo v koren, ali pa uporabimo lomljenje od zgoraj navzdol (glej razdelek 2.1.2), kjer izvajamo lomljenje že med iskanjem ustreznega vozlišča [9].

2.1.1 Lomljenje od spodaj navzgor

Če gledamo pot, ki jo opravimo od korena do izbranega vozlišča, potem vsakič, ko se po poti premaknemo v levo, tej potezi pravimo CIK, in vsakič, ko se po tej poti premaknemo v desno, pravimo CAK. Nadalje, premiku za dva nivoja v levo pravimo CIK-CIK in v desno CAK-CAK. Če se najprej premaknemo v levo in nato v desno, potem tej potezi pravimo CIK-CAK, če pa se najprej premaknemo v desno in nato v levo, pa tej potezi pravimo CAK-CIK. To so edini primeri, kako se lahko premaknemo za dva nivoja po dostopni poti (t.j. pot od korena do izbranega vozlišča). Če je ta pot liha (t.j. liho število povezav od korena do izbranega vozlišča), potem bo na koncu potreben še en korak, bodisi CIK, bodisi CAK.

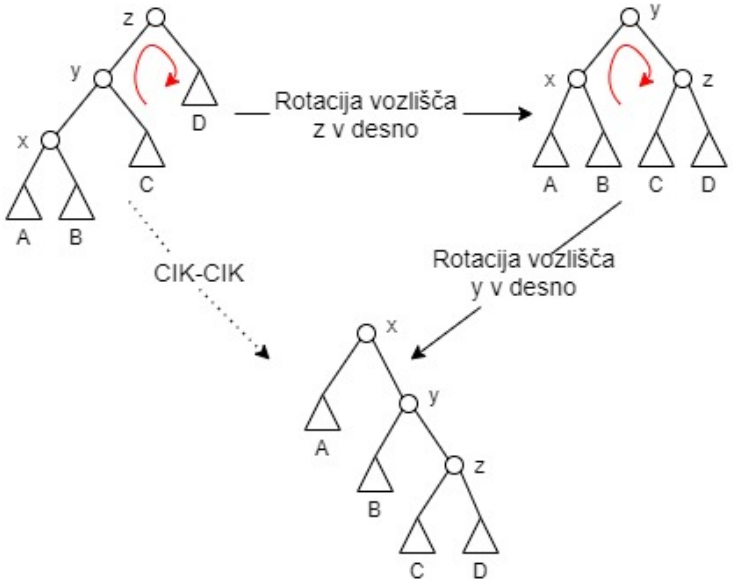
V nadaljevanju so opisane izvedbe rotacij CIK, CIK-CIK in CIK-CAK, ki se uporabljajo pri operaciji lomljenja. Opis izvedbe rotacij CAK, CAK-CAK in CAK-CIK izpustimo, saj so simetrične opisanim [9].

- CIK: če je y koren drevesa in starš vozlišča x, potem zavrtimo vozlišče y v desno, da vozlišče x postane novi koren drevesa. Ta primer je identičen enojni rotaciji. Primer rotacije CIK je prikazan na sliki 2.3.



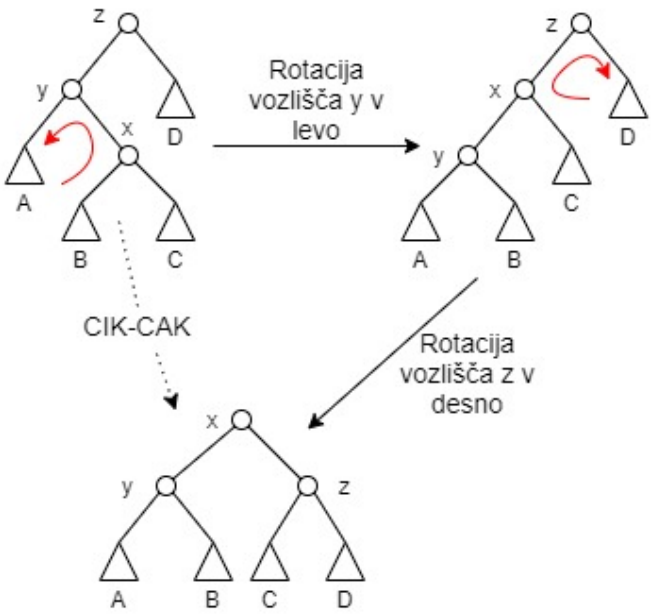
Slika 2.3: Premik vozlišča x v koren drevesa z rotacijo CIK.

- CIK-CIK: naj bo x vozlišče, ki ga želimo premakniti v koren, vozlišče y starš vozlišča x in vozlišče z starš vozlišča y. Naj še velja, da sta x in y oba leva otroka. Najprej zavrtimo vozlišče z v desno in nato zavrtimo še vozlišče y v desno, kot je prikazano na sliki 2.4.



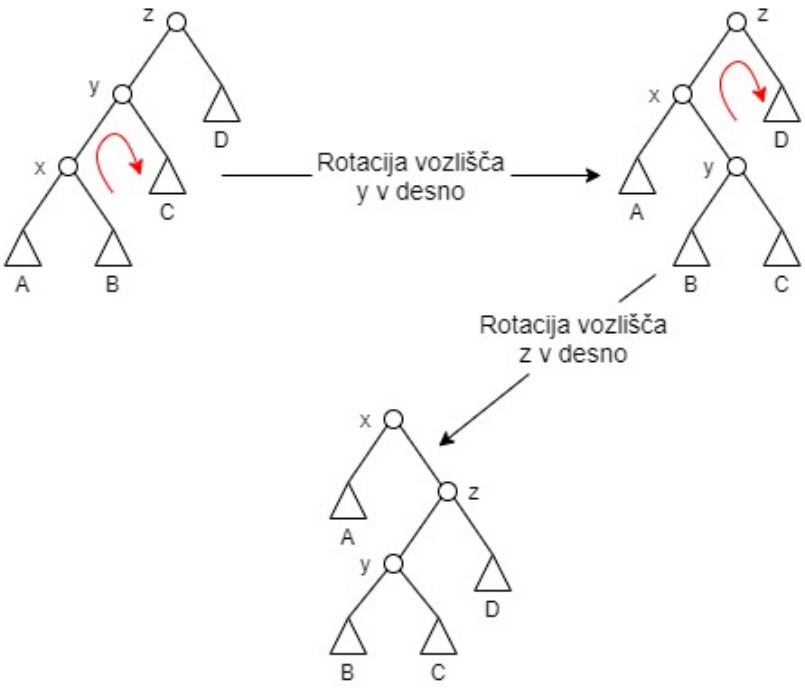
Slika 2.4: Premik vozlišča x v koren drevesa z rotacijo CIK-CIK.

- CIK-CAK: naj bo x vozlišče, ki ga želimo premakniti v koren in vozlišče y njegov starš ter y ni koren drevesa. Naj še velja, da je x desni otrok od y in y levi otrok od njegovega starša (vozlišča z). Najprej zavrtimo vozlišče y v levo in nato zavrtimo vozlišče z v desno, kot je prikazano na sliki 2.5.



Slika 2.5: Premik vozlišča x v koren drevesa z rotacijo CIK-CAK.

Opomba 2.1 ([5]) *Opazimo lahko, da primer CIK-CIK ni enak uporabi enojnih rotacij za premik izbranega vozlišča v koren drevesa. Ko uporabimo enojno rotacijo za premik vozlišča x v koren, najprej zavrtimo vozlišče y v desno in nato zavrtimo še vozlišče z v desno. Pri uporabi rotacije CIK-CIK pa najprej zavrtimo vozlišče z v desno in šele nato zavrtimo vozlišče y v desno. Potek rotacije CIK-CIK je prikazan na sliki 2.4, uporaba enojne rotacije pa je prikazana na sliki 2.6. Opazimo, da imata obe drevesi vozlišče x v korenu, se pa razlikujeta v strukturi desnega poddrevesa.*

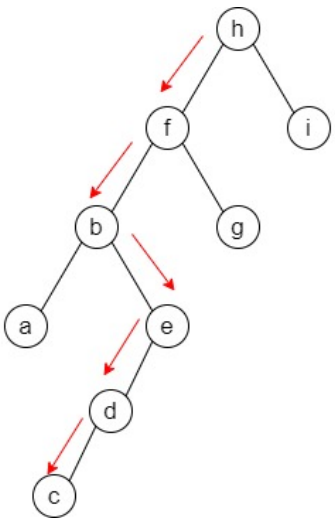


Slika 2.6: Premik vozlišča x v koren drevesa z uporabo enojnih rotacij.

Pri uporabi rotacij (CIK, CIK-CIK, CIK-CAK, itn.) pozicijo v drevesu menjajo samo vozlišča, ki so na poti do izbranega vozlišča.

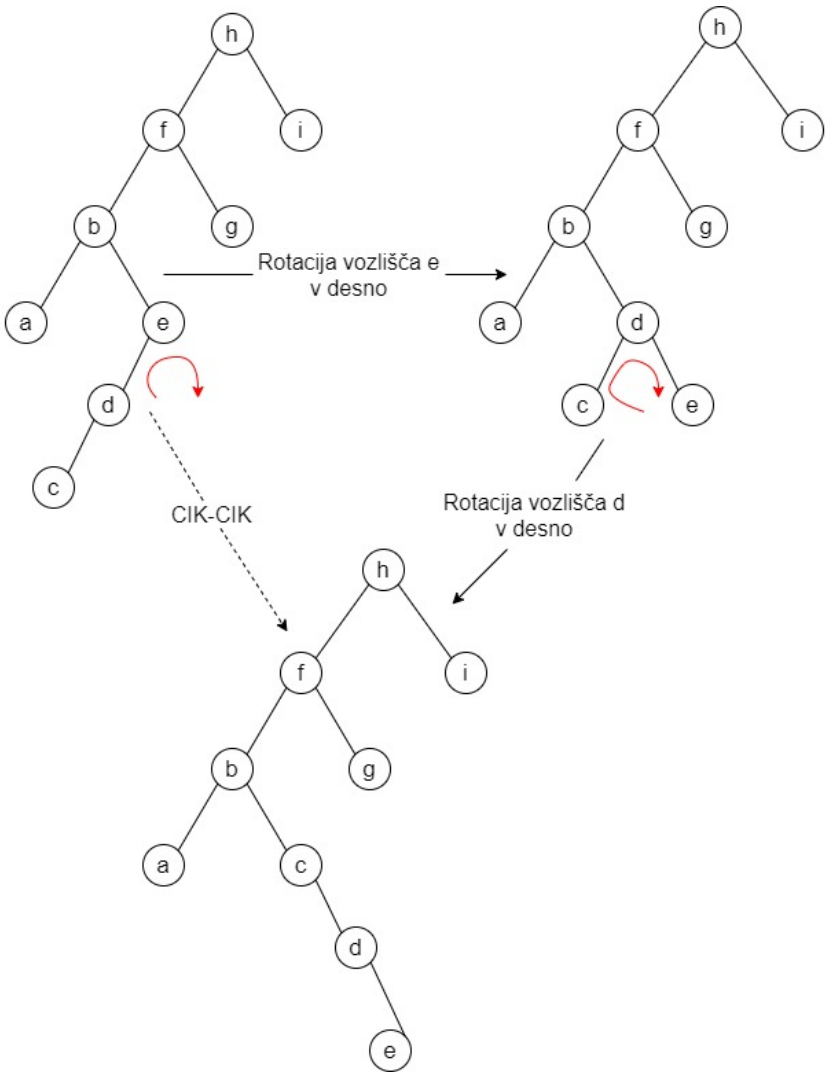
V naslednjem zgledu je prikazana uporaba operacije lomljenja na drevesu.

Zgled. Na sliki 2.7 imamo prikazano drevo, na katerem želimo izvesti operacijo lomljenja na vozlišču c.



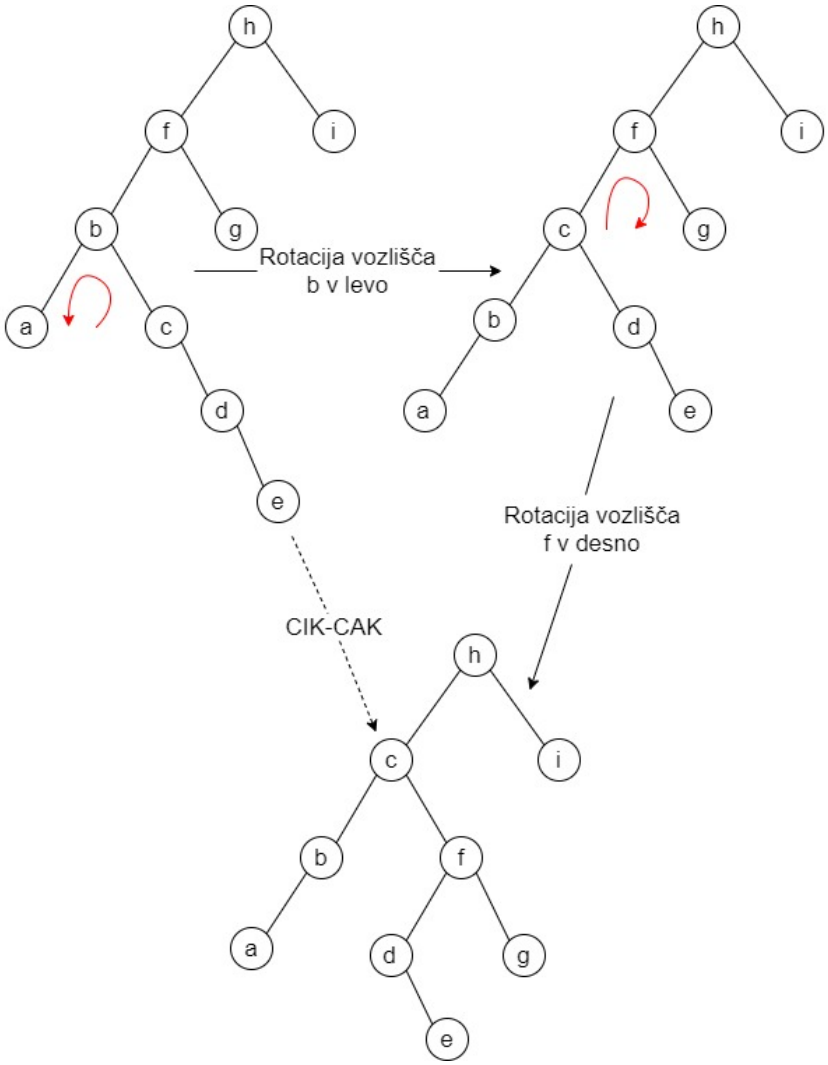
Slika 2.7: Začetno drevo, kjer želimo vozlišče c premakniti v koren drevesa.

Na poti od korena do vozlišča c so vozlišča h, f, b, e, d, c. Če gledamo po drevesu od spodaj navzgor od vozlišča c, potem na vozliščih c, d in e uporabimo rotacijo CIK-CIK, kjer položaj v drevesu spremenijo samo ta vozlišča. Ostalo drevo ostane nespremenjeno. Po uporabi rotacije CIK-CIK dobimo drevo, kot je prikazano na sliki 2.8.



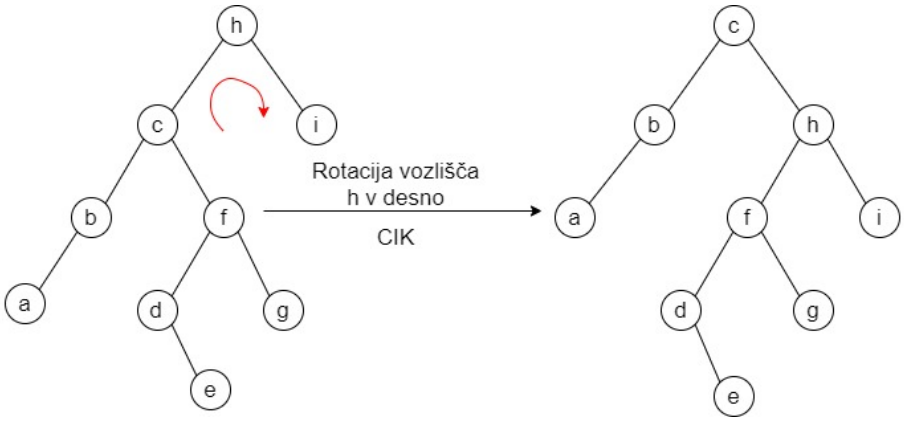
Slika 2.8: Prikaz izvedbe rotacije CIK-CIK z vmesnimi koraki.

Na vozliščih c, b in f uporabimo rotacijo CIK-CAK. Kot lahko vidimo na sliki 2.9, se poddrevo s korenom d ne spremeni, le premakne se na novo pozicijo.



Slika 2.9: Izvedemo rotacijo CIK-CAK in vozlišče c ponovno premaknemo dva nivoja višje.

Na koncu je vozlišče c levi otrok od korena, zato uporabimo še rotacijo CIK in dobimo končno drevo, kot je prikazano na sliki 2.10, kjer je c nov koren.



Slika 2.10: Na koncu izvedemo še rotacij CIK, da vozlišče c postane koren drevesa.

V prejšnjem zgledu smo uporabili lomljenje od spodaj navzgor, kjer začnemo pri vozlišču, ki ga želimo premakniti v koren drevesa. Predstavljeni način lomljenja je metoda, ki je sestavljena iz dveh operacij. Najprej poiščemo vozlišče v drevesu in nato najdeno vozlišče premaknemo v koren.

2.1.2 Lomljenje od zgoraj navzdol

Druga možnost izvedbe operacije lomljenja je lomljenje od zgoraj navzdol, kjer drevo lomimo že med samim iskanjem vozlišča z določenim ključem. Pri tem uporabljamo enake rotacije kot pri lomljenju od spodaj navzgor (CIK, CAK, CIK-CIK, CAK-CAK, CIK-CAK in CAK-CIK) [6].

Tekom izvajanja lomljenja prvotno drevo razpade na tri poddrevesa, ki se na koncu povežejo v novo drevo z iskanim ključem v korenu. Poddrevesa, ki jih skozi operacijo vzdržujemo so naslednja [5, 6]:

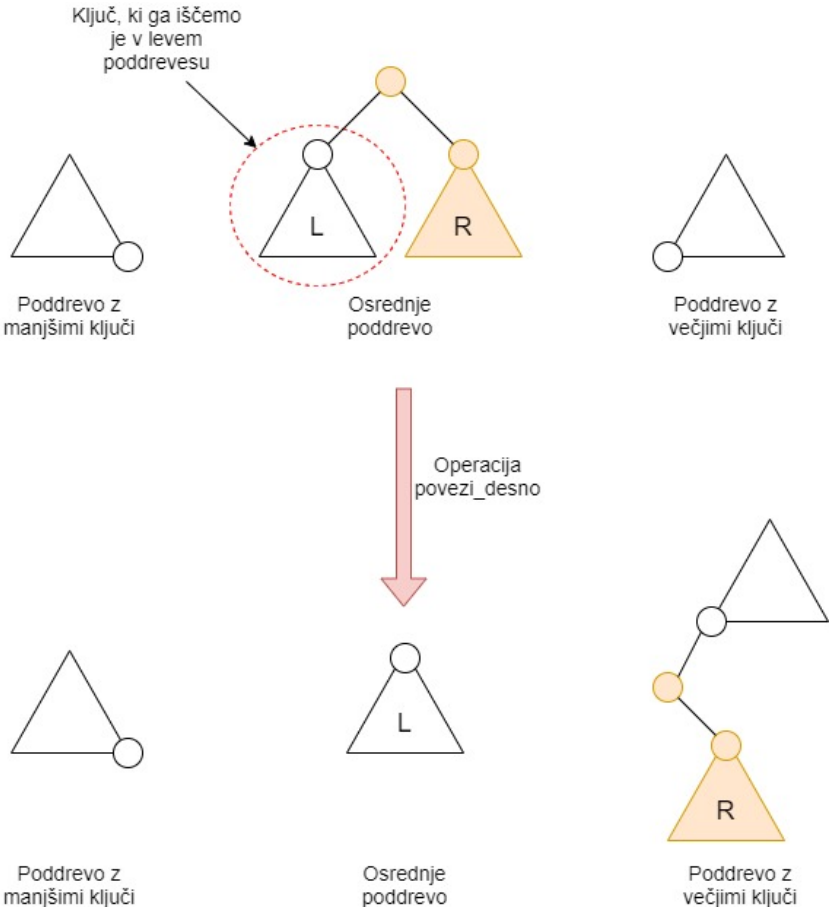
POGOJ 1 **osrednje poddrevo**: na začetku je to kar celotno drevo in vedno vsebuje poddrevesa, kjer pričakujemo, da vozlišče z iskanim ključem leži. Na vsakem koraku se število vozlišč v osrednjem poddrevesu zmanjša, dokler ne dosežemo iskanega ključa,

POGOJ 2 **poddrevo z manjšimi ključi**: vsebuje vozlišča s ključi, ki so manjša od ključa iskanega vozlišča oz. vsak ključ v poddrevesu z manjšimi ključi je manjši od vsakega ključa v osrednjem poddrevesu. Na začetku je poddrevo z manjšimi ključi prazno,

POGOJ 3 **poddrevo z večjimi ključi**: vsebuje vozlišča s ključi, ki so večja od ključa iskanega vozlišča oz. vsak ključ v poddrevesu z večjimi ključi je večji od vsakega ključa v osrednjem poddrevesu. Na začetku je poddrevo z večjimi ključi prazno.

Definicija 2.2 ([5]) *Pri lomljenju od zgoraj navzdol drevo razpade na poddrevo z manjšimi ključi, poddrevo z večjimi ključi in osrednje poddrevo. Pogojem, ki jim na vsakem koraku lomljenja zadoščajo poddrevesa, pravimo* trosmerna invarianta.

Na začetku je v osrednjem poddrevesu naše celotno drevo ter sta ostali poddrevesi prazni. Ker poddrevesa zadoščajo pogojem 1, 2 in 3 velja trosmerna invarianta. Na vsakem koraku iskanja v osrednjem poddrevesu primerjamo iskani ključ s ključem v korenu. Če je iskani ključ manjši od ključa v korenu, se iskanje premakne v levo poddrevo. Zato iz osrednjega poddrevesa odstranimo koren in njegovo desno poddrevo in ju vstavimo v poddrevo z večjimi ključi. Trosmerna invarianta pove, kam v poddrevo z večjimi ključi vstavimo koren in njegovo desno poddrevo. Ker je vsak ključ v osrednjem poddrevesu manjši od vseh ključev v poddrevesu z večjimi ključi, moramo koren (z njegovim desnim poddrevesom) vstaviti na levo stran skrajnega levega vozlišča v poddrevesu z večjimi ključi. Opisan postopek operacije imenujemo *povezi_desno*, ki je prikazan na sliki 2.11 [5, 6].



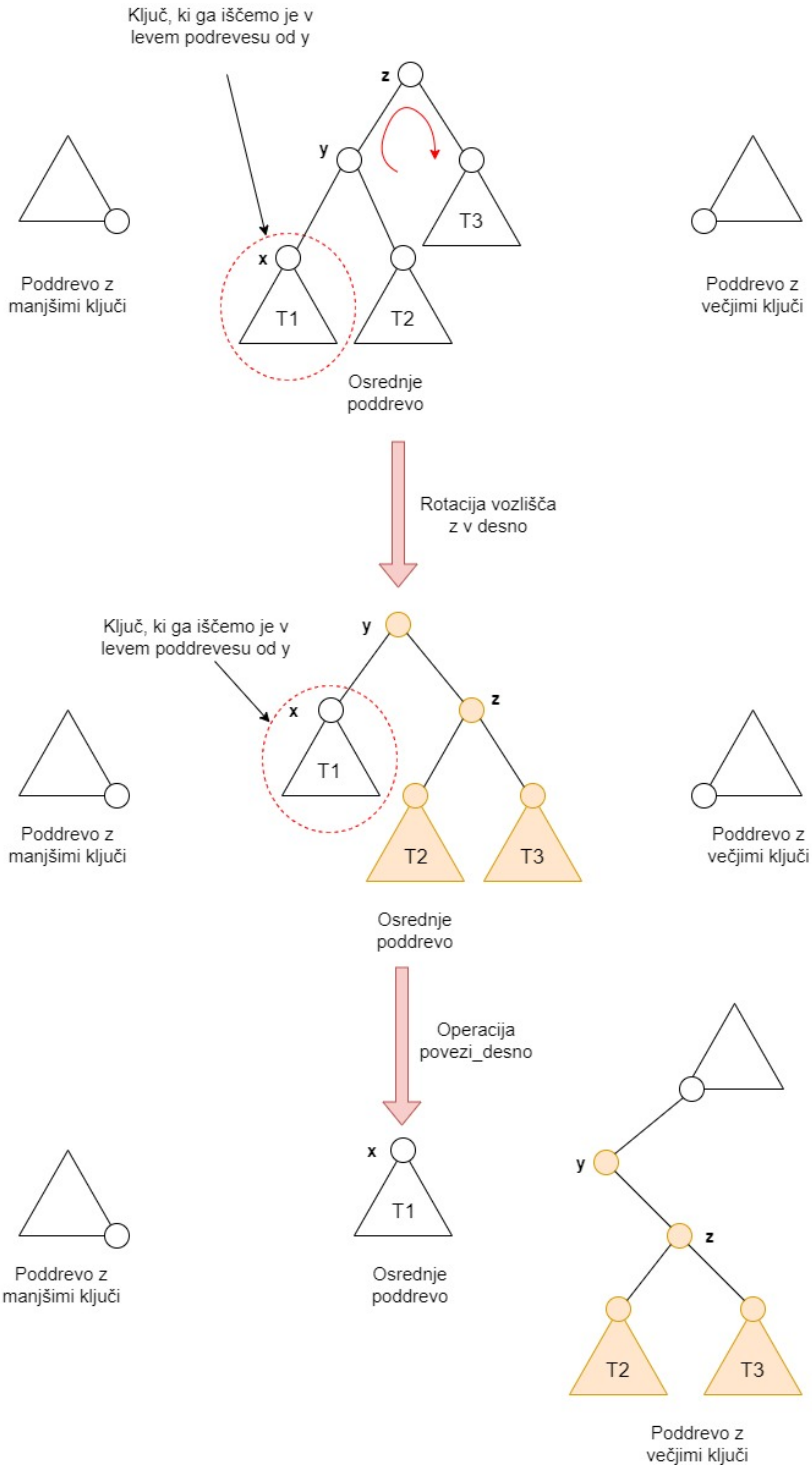
Slika 2.11: Prikaz delovanja operacije *povezi_desno*.

Analogno definiramo operacijo *povezi_levo*, kjer vozlišča odstranjujemo iz osrednjega poddrevesa in vstavljamo v poddrevo z manjšimi ključi.

Če primerjamo rotacijo CIK z operacijo *povezi_desno*, se v obeh primerih levi otrok korena premakne nivo višje in nadomesti svojega starša, ki se premakne nivo nižje v desno poddrevo. Dejansko je operacija *povezi_desno* ravno rotacija CIK le, da povezavo od nekdanjega levega otroka do korena izbrišemo. Namesto tega se koren (skupaj z desnim poddrevesom) premakne v drevo z večjimi ključi [5, 6].

Preostale rotacije CIK-CIK, CIK-CAK, CAK, itn. lahko izvedemo z uporabo operacij *povezi_desno*, *povezi_levo* in navadnimi (enojnimi) rotacijami v levo in desno.

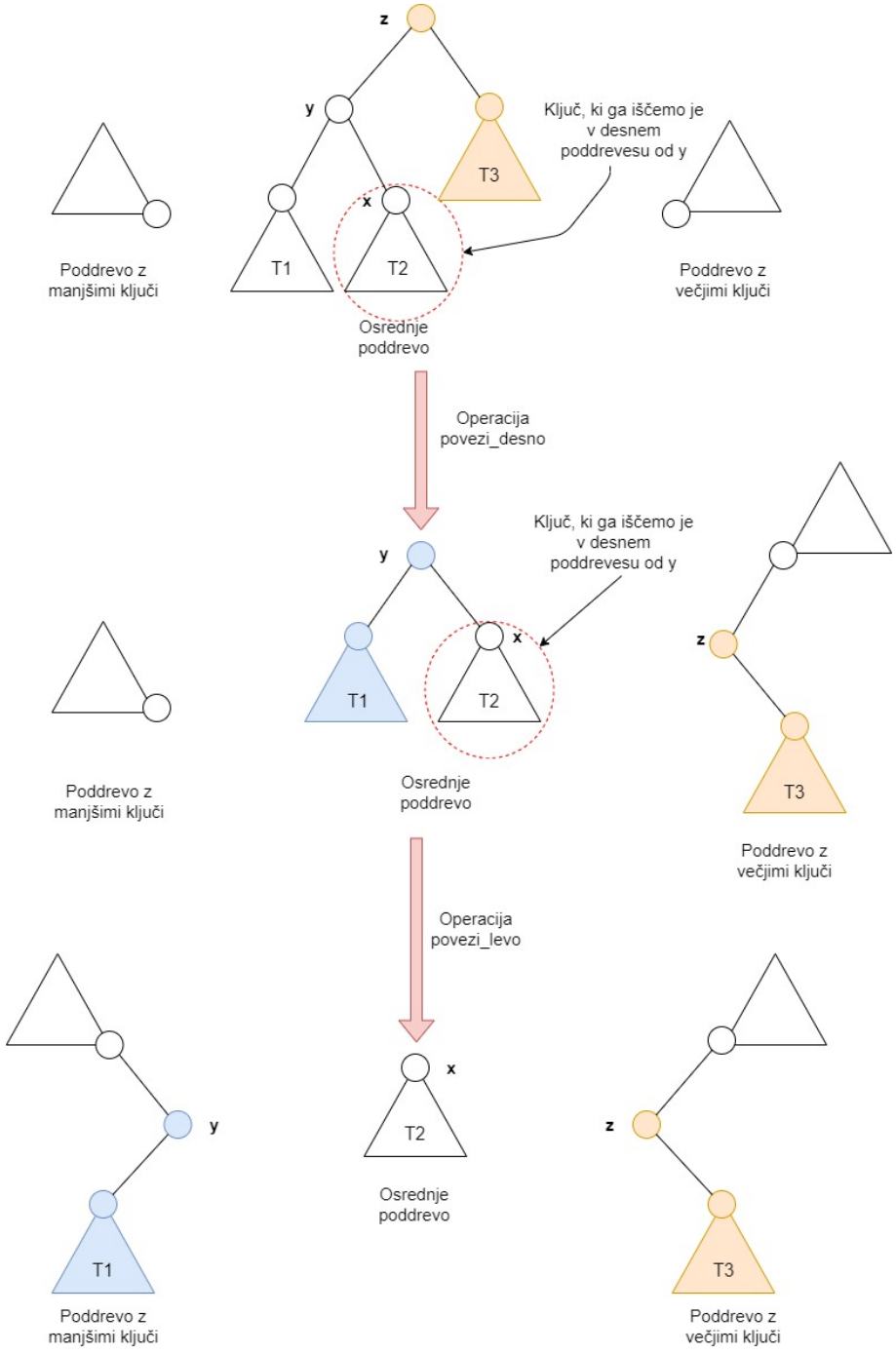
V primeru, da je iskani ključ manjši, tako od ključa v korenu, kot tudi od ključa levega otroka korena v osrednjem poddrevesu, moramo koren, njegovega levega otroka in njuni desni poddrevesi premakniti v poddrevo z večjimi ključi. Najprej koren osrednjega poddrevesa zavrtimo v desno in šele nato izvedemo operacijo *povezi_desno*. Opisan postopek je enak rotaciji CIK-CIK, ki je prikazan na sliki 2.12. Po končani izvedbi trosmerna invarianta še vedno velja [5].



Slika 2.12: Prikaz delovanja rotacije CIK-CIK.

V primeru CIK-CAK rotacije, prikazane na sliki 2.13, je iskani ključ nekje med korenom in desnim poddrevesom levega otroka od korena. Zato lahko koren in njegovo desno poddrevo z operacijo *povezi_desno* premaknemo v poddrevo z večjimi ključi ter levega otroka od korena

in njegovo levo poddrevo premaknemo z operacijo *povezi_levo* v poddrevo z manjšimi ključi. Ponovno po končani izvedbi trosmerna invarianta še vedno velja [5].

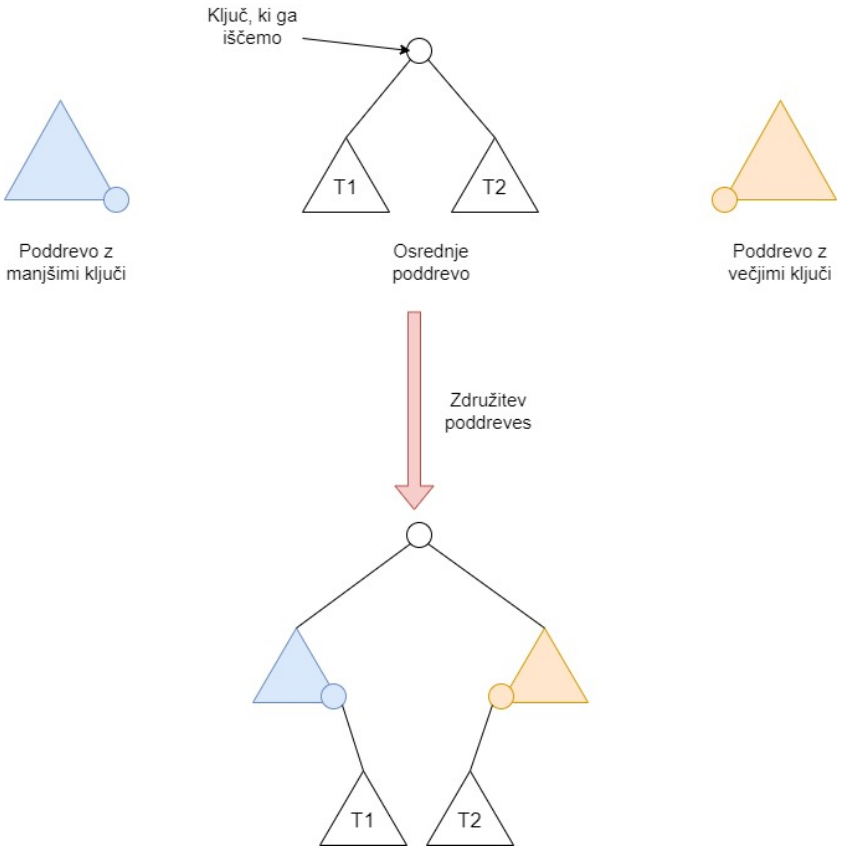


Slika 2.13: Prikaz delovanja rotacije CIK-CAK.

Lomljenje se konča, ko je iskani ključ v korenu osrednjega poddrevesa. Sledi postopek spajanja poddreves, ki je prikazan na sliki 2.14. Če ima koren osrednjega poddrevesa levo ali desno poddrevo (ali obe poddrevesi), potem levo poddrevo premaknemo v poddrevo

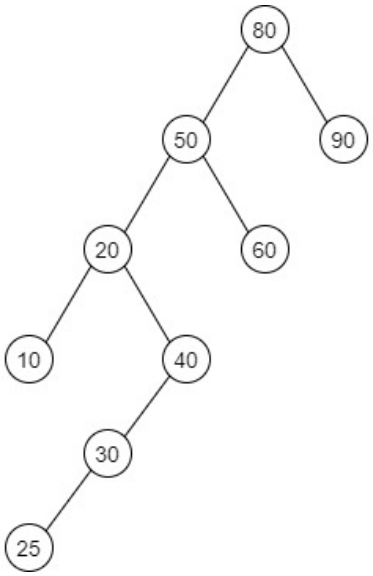
z manjšimi ključi, tako da levo poddrevo postane desni otrok skrajnega desnega vozlišča poddrevesa z manjšimi ključi, in desno poddrevo premaknemo v poddrevo z večjimi ključi, tako da desno poddrevo postane levi otrok skrajnega levega vozlišča drevesa z večjimi ključi. Nato spojimo poddrevesa skupaj. Poddrevo z manjšimi ključi postane levi otrok osrednjega poddrevesa in poddrevo z večjimi ključi postane desni otrok osrednjega poddrevesa [5, 6].

V primeru, da je osrednje poddrevo po lomljenju prazno, ustvarimo novo vozlišče z iskanim ključem. Ostali poddrevesi postaneta levi in desni otrok novega vozlišča.



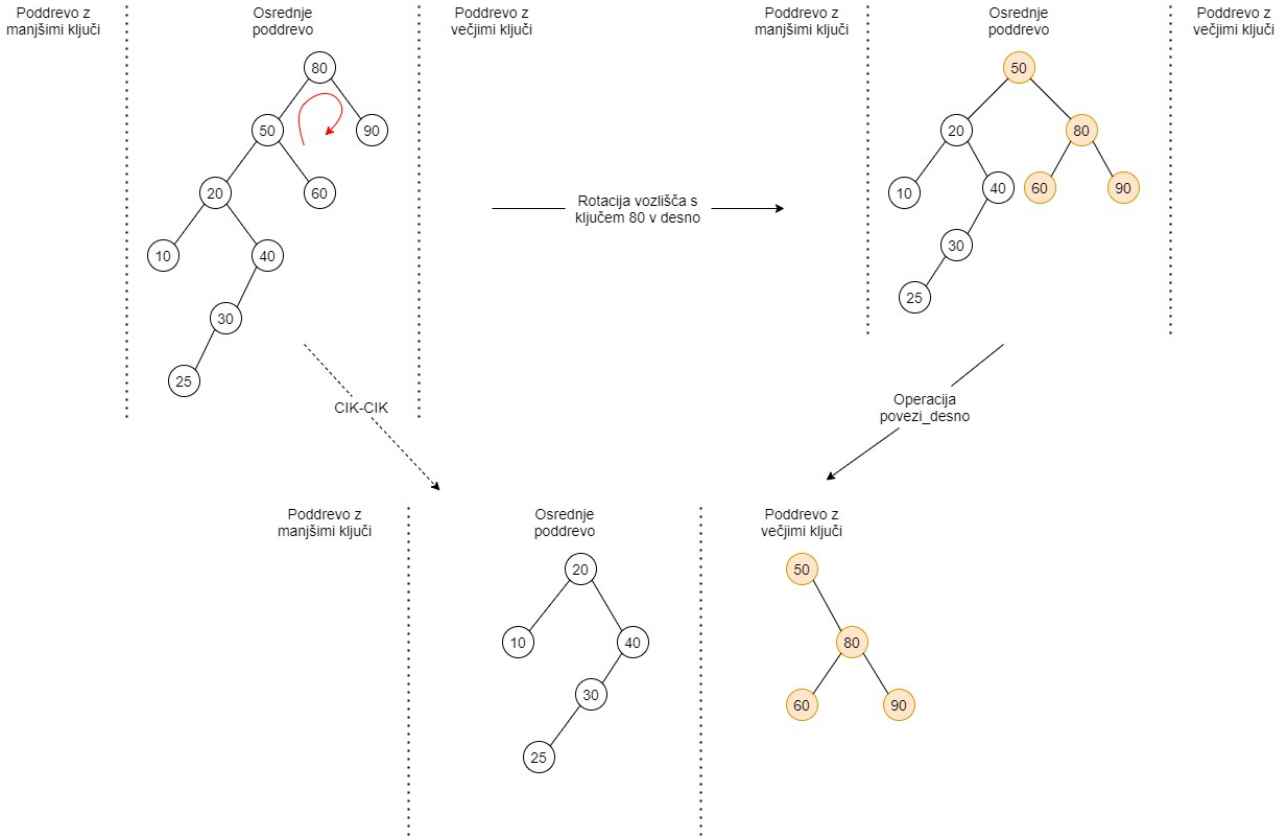
Slika 2.14: Prikaz spajanja poddreves.

Zgled. Imamo drevo na sliki 2.15, na katerem želimo izvesti operacijo lomljenja od zgoraj navzdol nad vozlišču s ključem 25.



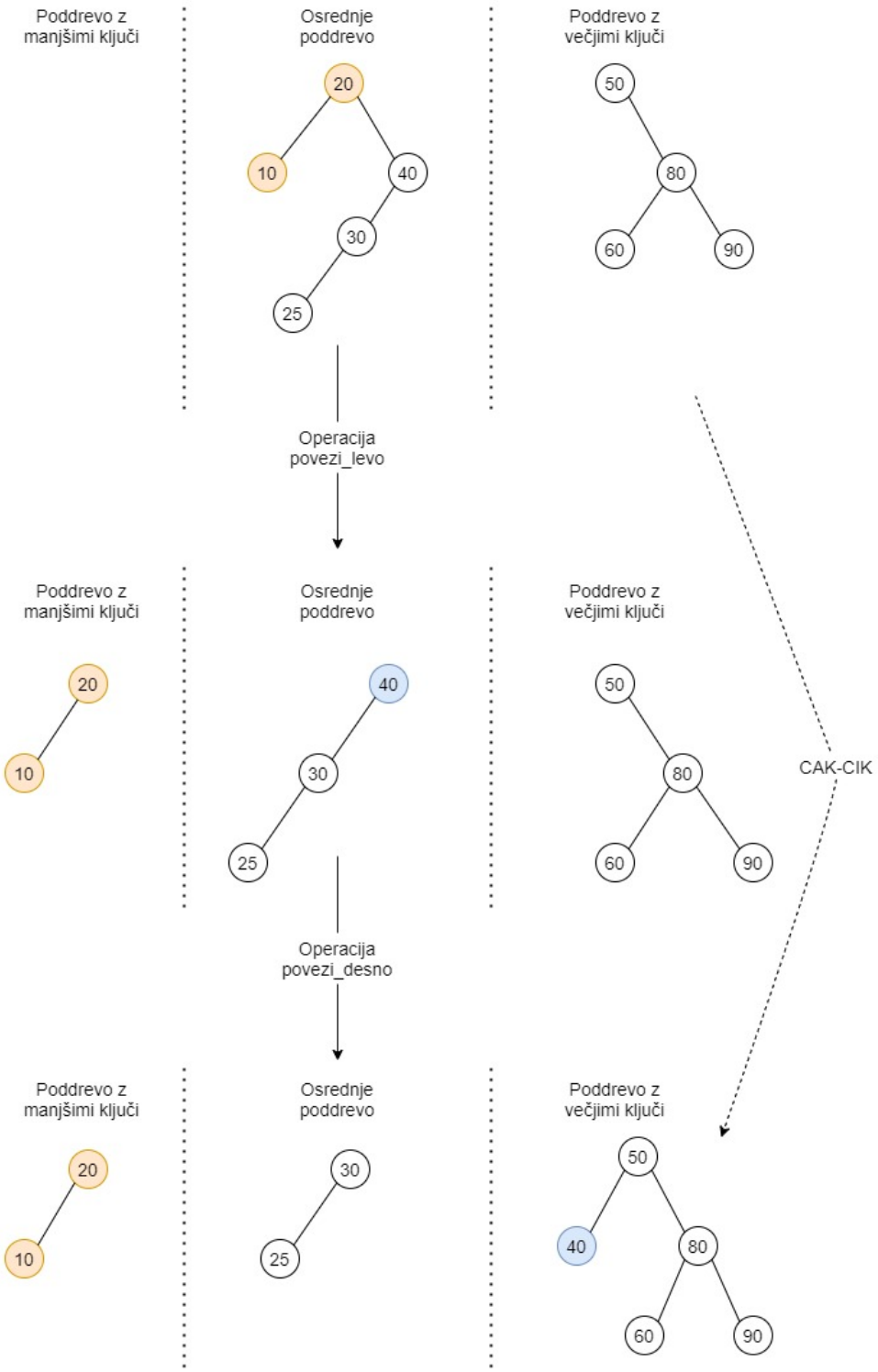
Slika 2.15: Začetno drevo na katerem želimo izvesti lomljenje od zgoraj navzdol.

Drevo tekom izvajanja operacije lomljenja razpade na tri poddrevesa. Na začetku je v osrednjem poddrevesu celotno drevo, poddrevo z manjšimi in poddrevo z večjimi ključi sta prazni. Kot vidimo na sliki 2.16 je 25 manjše od ključa v korenu in tudi manjše od ključa levega otroka korena, zato uporabimo rotacijo CIK-CIK. Najprej zavrtimo vozlišče s ključem 80 v desno ter nato novi koren osrednjega poddrevesa in njegovo desno poddrevo z operacijo *povezi_desno* premaknemo v poddrevo z večjimi ključi.



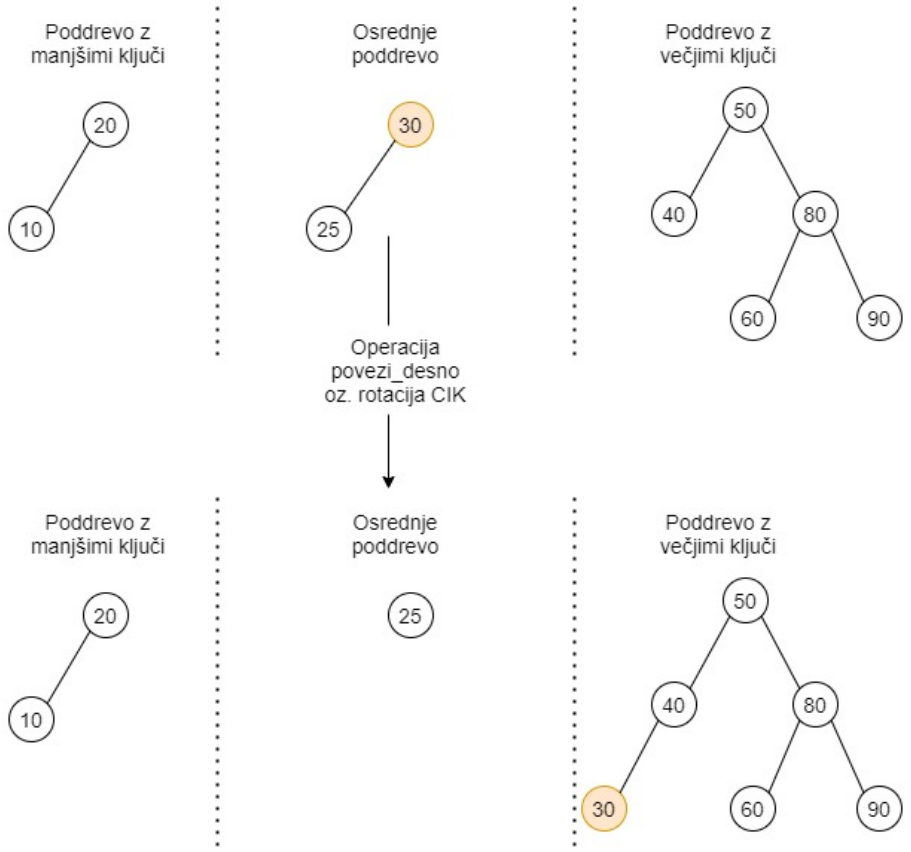
Slika 2.16: Prikaz postopka izvedbe rotacije CIK-CIK.

Ker je ključ 25 večji od ključa korena osrednjega poddrevesa in manjši od ključa desnega otroka korena, uporabimo rotacijo CAK-CIK. Kot prikazuje slika 2.17, najprej izvedemo operacijo *povezi_levo* in nato še operacijo *povezi_desno*.



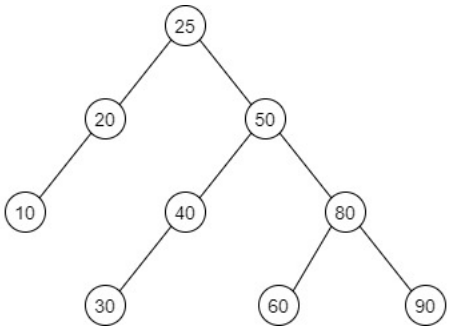
Slika 2.17: Prikaz postopka izvedbe rotacije CAK-CIK.

Ker je ključ 25 manjši od ključa v korenu osrednjega poddrevesa, uporabimo rotacijo CIK oz. operacijo *povezi_desno*, kot prikazuje slika 2.18.



Slika 2.18: Izvedemo še rotacijo CIK oz. operacijo $povezi_desno$.

V zadnjem koraku moramo poddrevesa še združiti skupaj. Ker je v osrednjem poddrevesu samo vozlišče s ključem 25, potem poddrevo z manjšimi ključi postane levo poddrevo osrednje poddrevesa ter poddrevo z večjimi ključi postane desno poddrevo osrednje poddrevesa. Slika 2.19 prikazuje končno drevo, kjer je vozlišče s ključem 25 koren.



Slika 2.19: Drevo, ki ga dobimo po izvedbi operacije lomljenja od zgoraj navzdol nad vozlišču s ključem 25.

2.2 Operacije na lomljenih drevesih

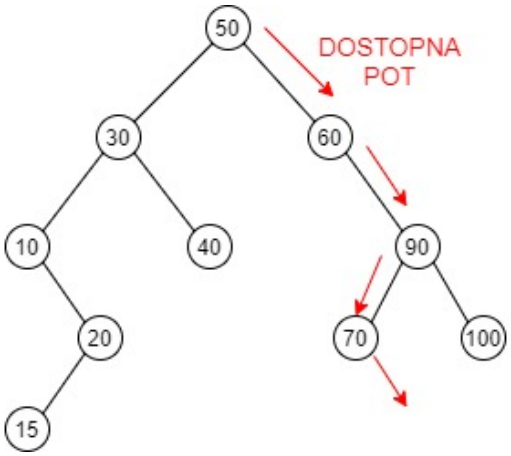
V poglavju 1.3.1 smo predstavili osnovne operacije na dvojiških iskalnih drevesih. V nadaljevanju je opisano delovanje teh operacij na lomljenih drevesih.

Opomba 2.3 *V operacijah iskanja, vstavljanja in odstranjevanja vedno uporabimo lomljenje od spodaj navzgor.*

2.2.1 Iskanje vozlišča z danim ključem

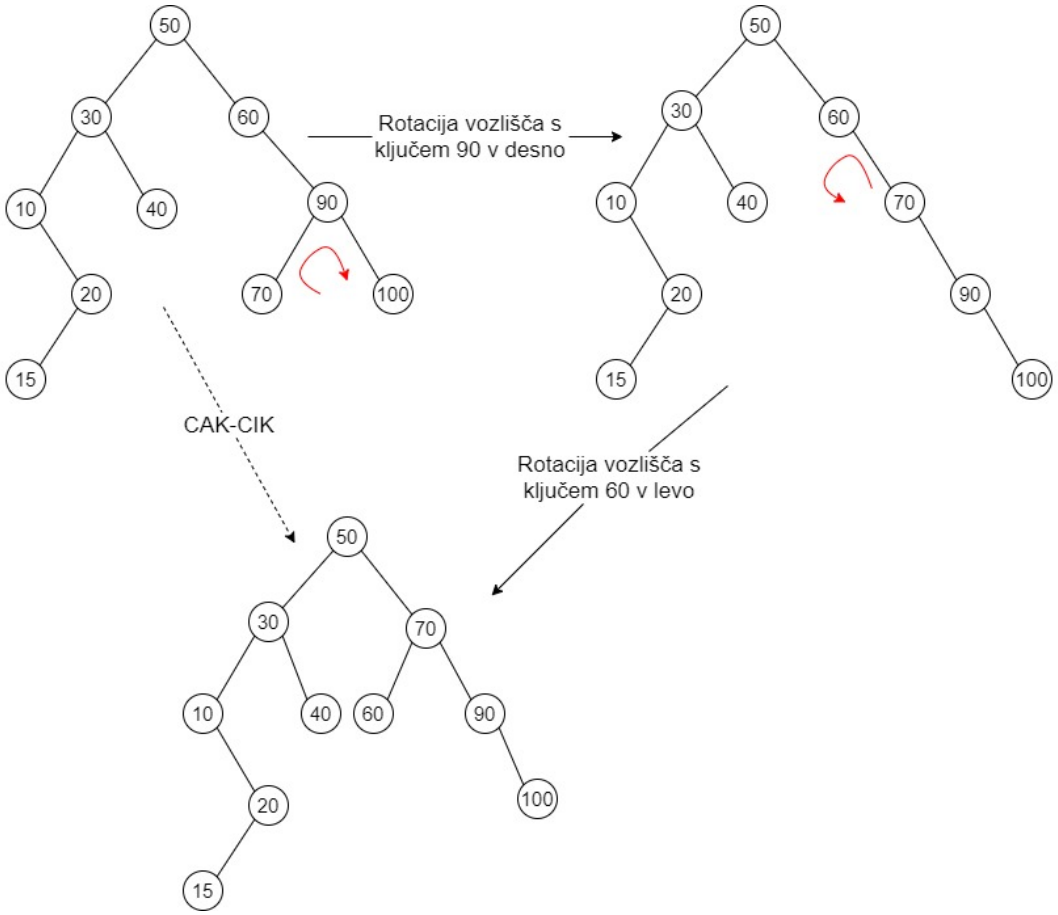
Na lomljenih drevesih operacija iskanje vozlišča z danim ključem i izvede iskanje od korena navzdol. Če je x najdeno vozlišče s ključem i, potem se izvede lomljenje na vozlišču x (x premaknemo v koren) in na koncu vrnemo kazalec na koren drevesa. Če ključa i ni v drevesu, potem uporabimo lomljenje na zadnjem neničelnem vozlišču in vrnemo ničelni kazalec (*nullptr*) [5, 9].

Zgled. Poiskati želimo vozlišče s ključem 80. Iskanje začnemo v korenu in se spuščamo po drevesu navzdol, kot je prikazano na sliki 2.20.



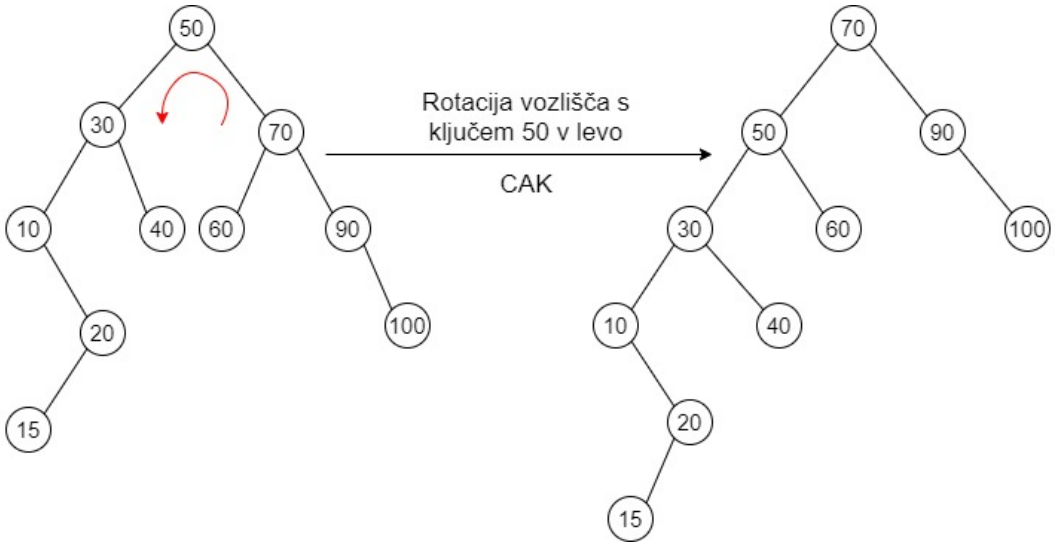
Slika 2.20: Prikazano je začetno drevo, v katerem želimo poiskati vozlišče s ključem 80.

Ker vozlišča s ključem 80 ni v drevesu, uporabimo lomljenje na zadnjem neničelnem vozlišču, torej na vozlišču s ključem 70. Najprej uporabimo rotacijo CAK-CIK, da vozlišče s ključem 70 premaknemo za dva nivoja višje, kot je prikazano na sliki 2.21.



Slika 2.21: Izvedba rotacije CAK-CIK po korakih.

Ker je vozlišče s ključem 70 desni otrok od vozlišča s ključem 50, moramo izvesti samo še rotacijo CAK, da bo to vozlišče koren drevesa. Zadnji korak lomljenja je prikazan na sliki 2.22.

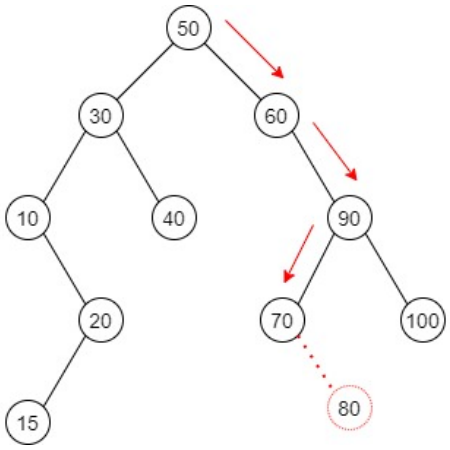


Slika 2.22: Zadnji korak lomljenja nad vozlišče s ključem 70 je izvedba rotacije CAK.

2.2.2 Vstavi vozlišče z danim ključem

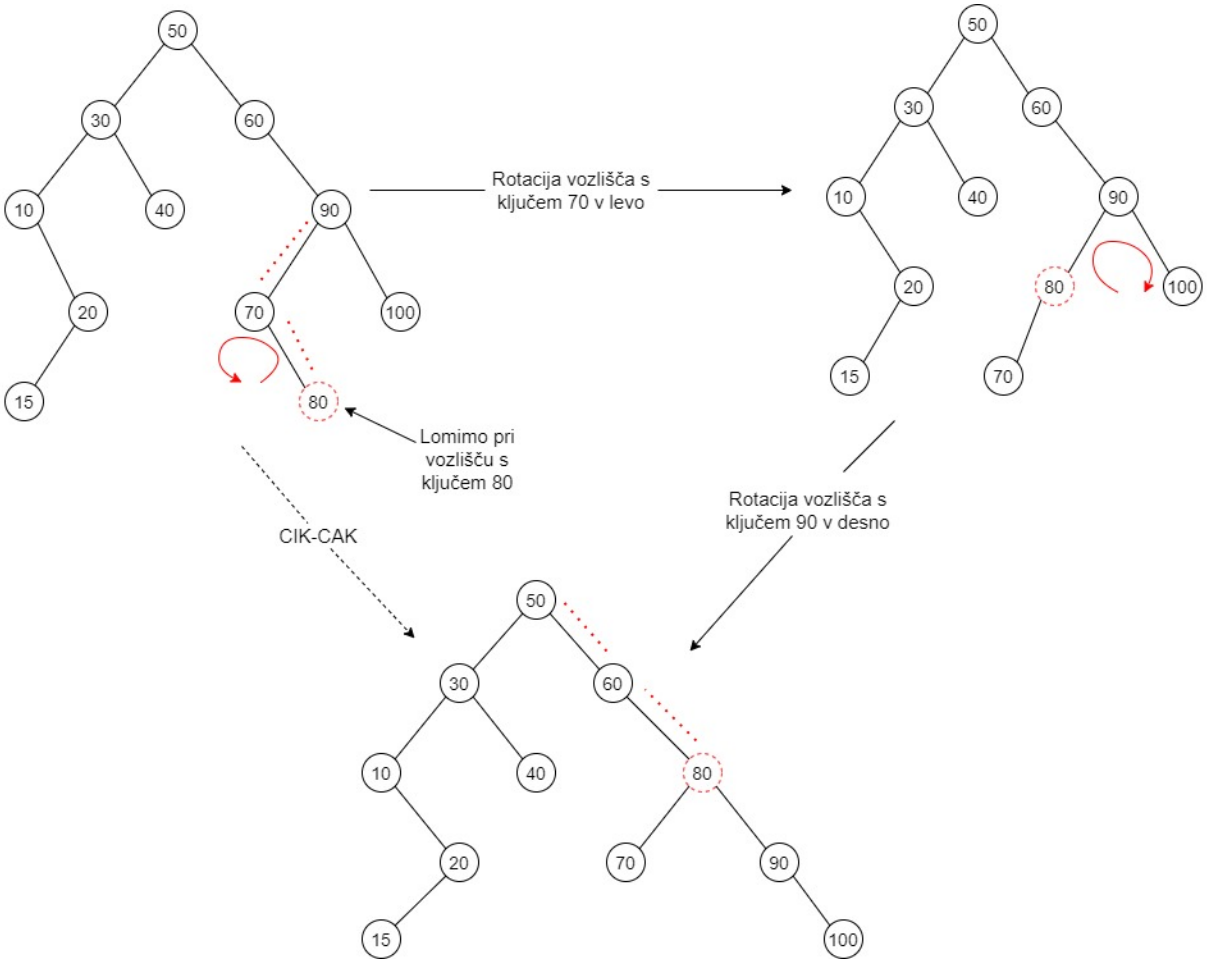
Na lomljenih drevesih najprej izvedemo običajno operacijo vstavljanja vozlišča z danim ključem v drevo, torej novo vozlišče postane list drevesa. Po vstavljanju se nato izvede operacija lomljenje nad novo vstavljenim vozliščem. Tako je na koncu izvedbe novo vozlišče koren lomljenega drevesa [5, 9].

Zgled. V drevo želimo vstaviti vozlišče s ključem 80. Najprej novo vozlišče postane desni otrok vozlišča s ključem 70, kot je prikazano na sliki 2.23.



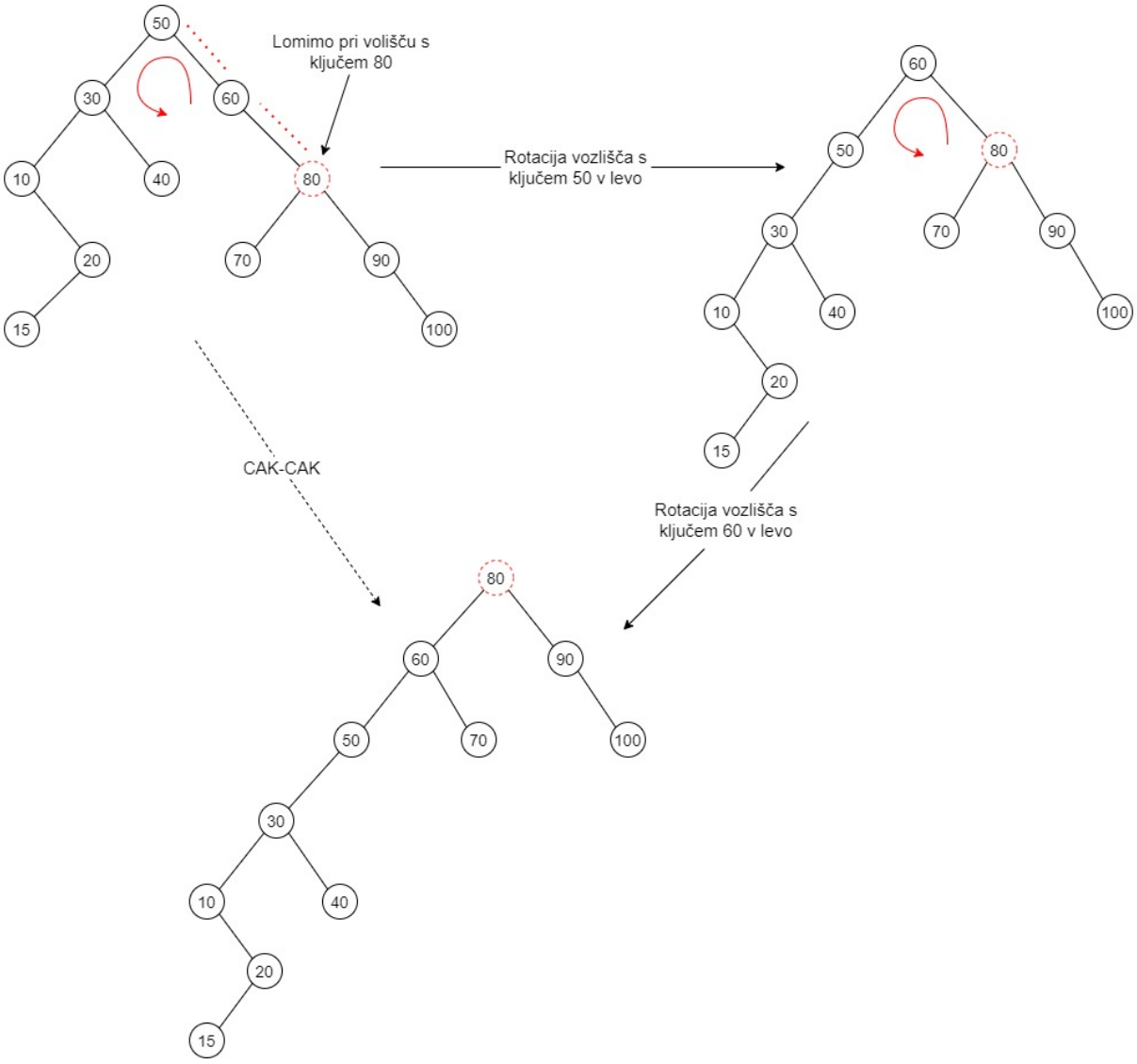
Slika 2.23: Vozlišče s ključem 80 postane desni otrok vozlišča s ključem 70.

Sledi operacija lomljenja, kjer najprej izvedemo rotacijo CIK-CAK. Njen potek je prikazan na sliki 2.24.



Slika 2.24: Prikaz izvedbe rotacije CIK-CAK.

Da bo vozlišče s ključem 80 v korenu, moramo izvesti še rotacijo CAK-CAK, kot prikazuje slika 2.25.



Slika 2.25: Prikaz izvedbe rotacije CAK-CAK.

2.2.3 Odstrani vozlišče z danim ključem

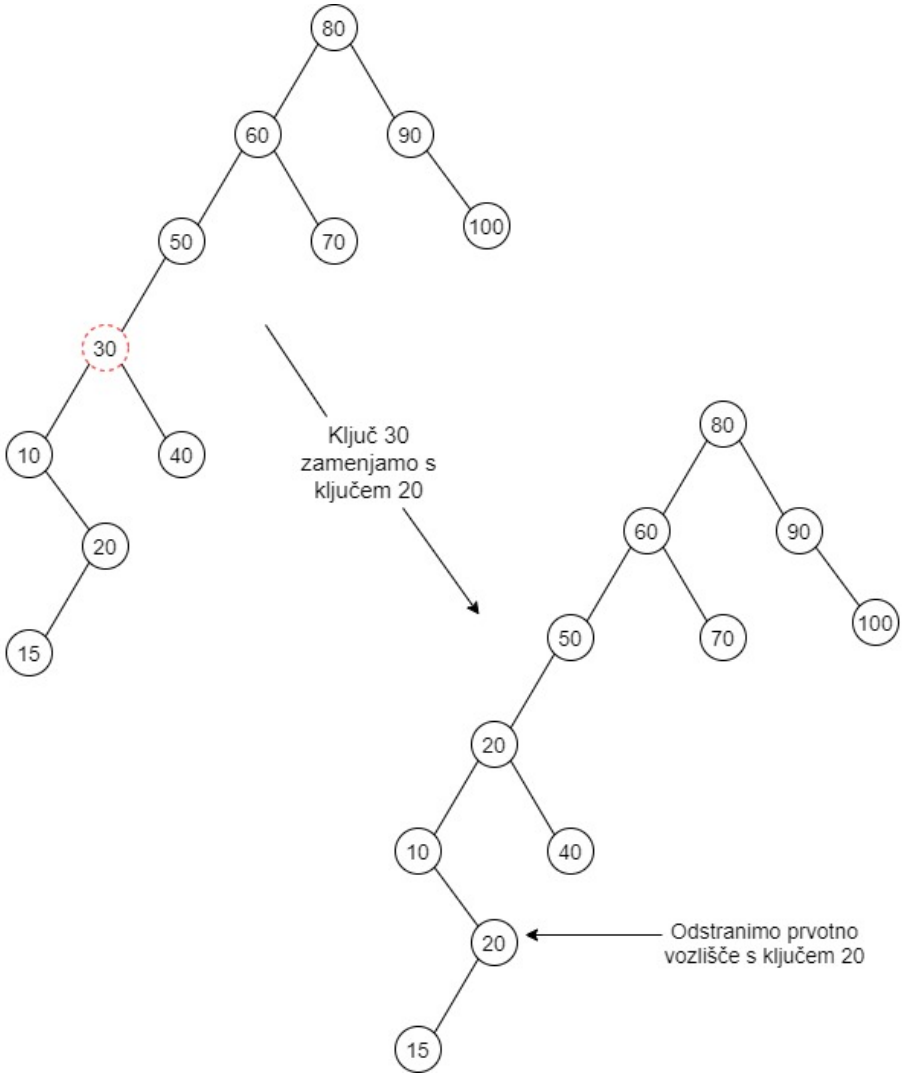
Naj bo x vozlišče s ključem i, ki ga želimo odstraniti iz lomljenega drevesa in $p(x)$ njegov starš. Pri odstranjevanju imamo naslednje možnosti [5, 9]:

1. če x nima otrok, potem kazalec od $p(x)$, ki je kazal na x, preusmerimo, da kaže na ničelno vozlišče in izvedemo lomljenje na vozlišču $p(x)$,
2. če ima x enega otroka, potem kazalec od $p(x)$, ki je kazal na x, preusmerimo, da kaže na edinega otroka od x in nato izvedemo lomljenje na vozlišču $p(x)$,
3. če ima x oba otroka, potem v levem poddrevesu od x poiščemo vozlišče y z največjim

ključem in x nadomestimo z vozliščem y. Prvotno vozlišče y odstranimo in izvedemo lomljenje na vozlišču $p(x)$,

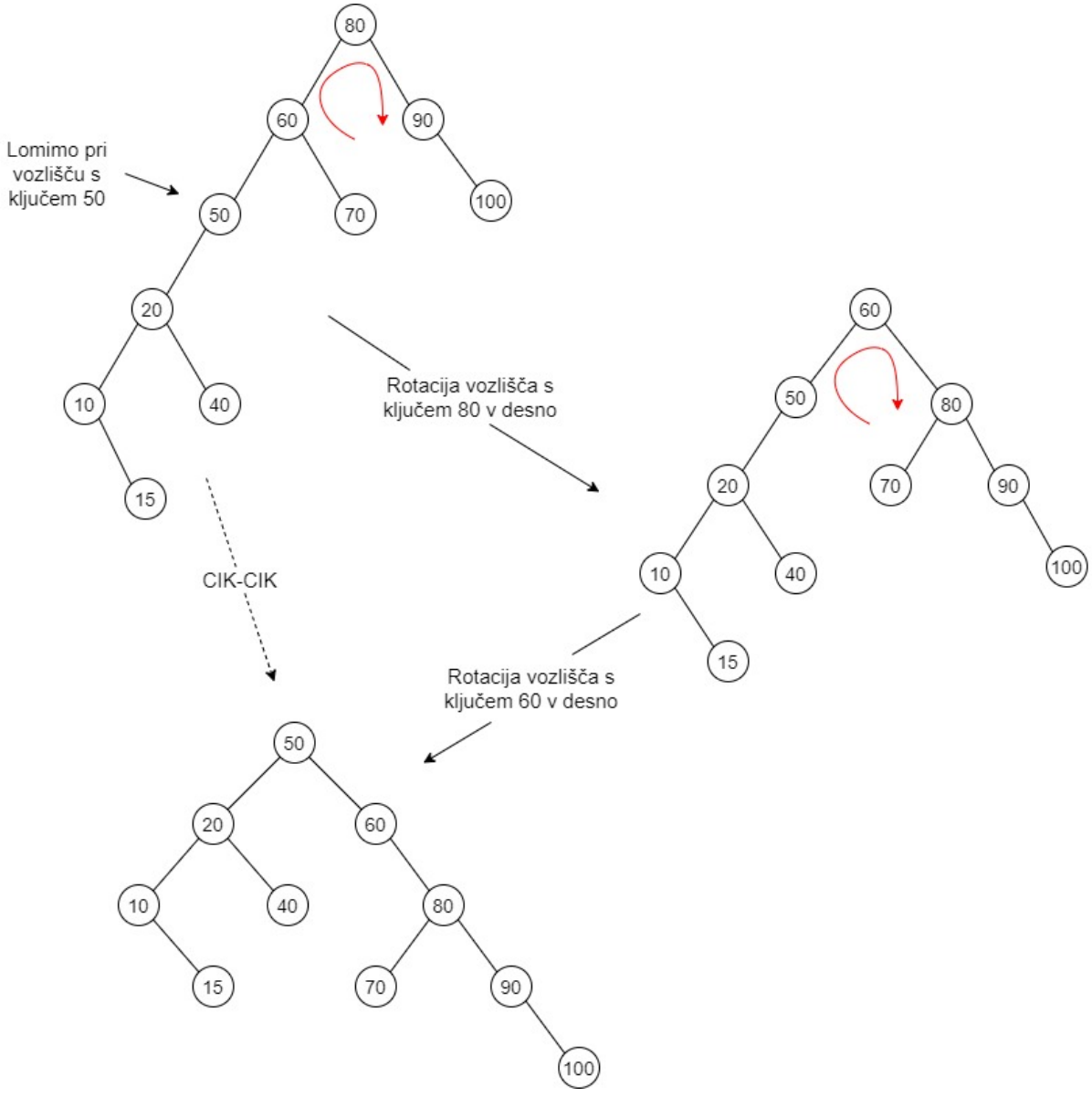
4. če vozlišča x s ključem i ni v drevesu, potem izvedemo lomljenje na zadnjem neničelnem vozlišču (t.j. zadnje neničelno vozlišče, kjer pričakujemo, vozlišče x).

Zgled. Iz drevesa želimo odstraniti vozlišče s ključem 30. To vozlišče ima oba otroka, zato najprej to vozlišče nadomestimo s ključem 20, kar nam prikazuje slika 2.26.



Slika 2.26: Vozišče s ključem 30 nadomestimo s ključem 20 in odstranimo prvotno vozlišče s ključem 20.

Prvotno vozlišče s ključem 20 odstranimo (z drugimi besedami, odstranimo vozlišče s ključem 30 in združimo levo in desno poddrevo). Na koncu sledi še operacija lomljenja. Lomimo pri vozlišču s ključem 50 (starš odstranjenega vozlišča). Pri tem uporabimo rotacijo CIK-CIK. Izvedbo operacije prikazuje slika 2.27.



Slika 2.27: Lomimo pri vozlišču s ključem 50, pri čemer uporabimo rotacijo CIK-CIK.

Poglavje 3

Amortizirana analiza zahtevnosti

Amortizirano analizo zahtevnosti uporabljamo za algoritme, kjer so nekatere operacije zelo počasne, vendar je večina drugih hitrejša. Običajno pri analizi algoritmov uporabimo analizo v najslabšem primeru in analizo povprečnega primera. V obeh primerih vzamemo eno izvedbo operacije in poskušamo določiti, kako zahtevna je. Amortizirana analiza se od ostalih razlikuje po tem, da namesto ene izvedbe operacije vzame zaporedje operacij. Amortizirana analiza tako poda oceno v najslabšem primeru daljšega zaporedja operacij [1, 2].

Primeri podatkovnih struktur, katerih operacije se analizirajo z amortizirano analizo zahtevnosti, so tabele simbolov, disjunktne množice (angl. disjoint sets) in lomljena drevesa.

V nadaljevanju so predstavljene tri najbolj uporabljene metode za analizo amortizirane zahtevnosti. To so agregatna, računovodska in metoda potencialov.

3.1 Agregatna metoda

Pri agregatni metodi predpostavimo, da ni potrebno razlikovati med različnimi operacijami nad podatkovno strukturo. Zato sta pri izračunu potrebna samo dva koraka:

1. pokažemo, da zaporedje n operacij v najslabšem primeru zahteva $T(n)$ časa in
2. pokažemo, da vsaka operacija v zaporedju zahteva $\frac{T(n)}{n}$ časa.

Zato ima pri agregatni metodi vsaka operacija enako zahtevnost [11].

Zgled. Vzemimo podatkovno strukturo sklad, ki deluje po principu $LIFO$ (angl. last in, first out). Zanima nas amortizirana zahtevnost operacij na skladu. Operaciji, ki ju uporabimo sta:

- $push(S, x)$: vstavi element x na vrh sklada S in
- $pop(S, x)$: odstrani element, ki je na vrhu sklada S in vrne odstranjeni element. Če je sklad prazen, pride do napake.

Operaciji *push* in *pop* imata zahtevnost $O(1)$, zato je skupna zahtevnost zaporedja n operacij *push* in *pop* $\Theta(n)$.

Dodajmo še operacijo $multipop(S, k)$, ki odstrani k elementov iz vrha sklada S, oziroma odstrani vse elemente, če je velikost sklada manjša od k. Pri implementaciji operacije $multipop(S, k)$ uporabimo *while* zanko, ki jo izvajamo tako dolgo dokler sklad ni prazen in je število $k > 0$ (k v tem primeru predstavlja število elementov, ki jih še moramo odstraniti s sklada). V telesu zanke najprej uporabimo operacijo $pop(S)$, ki odstrani element na vrhu sklada S, nato število k zmanjšamo za 1. V psevdokodi 3.1 funkcija *prazen_sklad* vrne $TRUE$, če je sklad prazen in $FALSE$ sicer.

```
void multipop(sklad S, int k){
    while(!prazen_sklad(S) && k>0){
        pop(S);
        k = k-1;
    }
}
```

Algoritem 3.1: Psevdokoda operacije multipop.

V operaciji *multipop* se *while* zanka izvede $\min\{s, k\}$ - krat, kjer je s velikost sklada. V vsaki ponovitvi zanke se izvede ena operacija *pop*, zato je skupna zahtevnost operacije *multipop* enaka $\min\{s, k\}$.

Če je velikost sklada n in izvedemo operacijo $multipop(S, n)$, potem je časovna zahtevnost v najslabšem primeru enaka $O(n)$. Če izvedemo n operacij $multipop(S, n)$ s časovno zahtevnostjo $O(n)$, potem je skupna časovna zahtevnost operacij enaka $O(n^2)$.

Recimo, da imamo prazen sklad, na katerem želimo izvesti poljubno zaporedje n operacij *push*, *pop* in *multipop*, potem je lahko časovna zahtevnost zaporedja operacij največ $O(n)$, saj lahko vsak element s sklada odstranimo največ enkrat, po tem ko smo ga vstavili na vrh sklada. Zato je za poljubno zaporedje n operacij *push*, *pop* in *multipop* časovna zahtevnost enaka $O(n)$.

Z uporabo agregatne metode je povprečna zahtevnost oz. amortizirana zahtevnost katerekoli operacije enaka $\frac{O(n)}{n} = O(1)$.

3.2 Računovodska metoda

Za razliko od agregatne metode, računovodska metoda vsaki vrsti operacije dodeli različne stroške. Računovodska metoda je zelo podobna upravljanju osebnih financ: stroške svojega poslovanja lahko ocenimo kakorkoli želimo, če na koncu dneva znesek denarja, ki ga bomo dali na stran, zadostuje za plačilo računov. Ocenjeni stroški operacije so lahko večji ali manjši od dejanskih stroškov; ustrezno se lahko presežek ene operacije uporabi za plačilo dolga drugih operacij [1].

Zapišimo to s simboli. Dani operaciji, katere dejanski strošek je c, dodelimo amortiziran strošek $\widehat{c}$. Amortizirani stroški morajo zadoščati pogoju, da za poljubno zaporedje n operacij z dejanskimi stroški $c_1, c_2, \ldots, c_n$ in amortiziranimi stroški $\widehat{c_1}, \widehat{c_2}, \ldots, \widehat{c_n}$ velja

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i.$$

Dokler ta neenakost velja, vemo, da je amortizirani strošek zgornja meja dejanskih stroškov poljubnega zaporedja operacij. Razlika med zgornjima vsotama je skupni presežek ali dobropis, shranjen v podatkovni strukturi, ki mora biti vedno nenegativen [1, 10].

Zgled. Poglejmo računovodsko metodo na podatkovni strukturi sklad. Operacijam na skladu lahko dodelimo dejanske in amortizirane stroške:

Operacija	Dejasnki strošek c_i	Amortizirani strošek $\widehat{c_i}$
push	1	2
pop	1	0
multipop	$\min\{\|S\|, k\}$	0

Tabela 3.1: V tabeli so zapisani dejanski in amortizirani stroški operacij na skladu.

Ko element vstavimo v sklad, dobimo dovolj dobropisa za plačilo, ne le za operacijo vstavljanja v sklad, temveč tudi za operacijo, ki bo element odstranila iz sklada, ne glede na to ali je operacija *pop*, *multipop* ali sploh brez operacije.

3.3 Metoda potencialov

Naj bo D_0 začetna podatkovna struktura. Ko na podatkovni strukturi D_0 izvedemo poljubno operacijo (npr. iskanje, vstavljanje, odstranjevanje vozlišča) dobimo novo podatkovno strukturo D_1. Če na podatkovni strukturi D_1 izvedemo poljubno operacijo, dobimo novo podatkovno strukturo D_2. Če postopek ponavljamo, potem po izvedbi i-te operacije na podatkovni strukturi D_{i-1} dobimo novo podatkovno strukturo D_i.

Pri metodi potencialov izvedemo n operacij, pri čemer začnemo z začetno podatkovno strukturo D_0. Za vsak $i \in \{1, 2, \ldots, n\}$, naj bo c_i dejanska zahtevnost i-te operacije in naj bo D_i podatkovna struktura, ki nastane po uporabi i-te operacije na podatkovni strukturi D_{i-1}. Potencialna funkcija Φ preslika vsako podatkovno strukturo D_i v realno število $\Phi(D_i)$, kar predstavlja *potencial* povezan s podatkovno strukturo D_i. Amortizirana zahtevnost $\widehat{c_i}$ i-te operacije glede na potencialno funkcijo Φ [1] je definirana z

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{3.1}$$

Amortizirana zahtevnost vsake operacije je potem enaka dejanski zahtevnosti plus spremembi potencialov zaradi operacije. Po enačbi 3.1 je skupna amortizirana zahtevnost n operacij [1] enaka

$$\begin{aligned} \sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \end{aligned} \tag{3.2}$$

Za nadaljno obravnavo lahko to dejstvo zapišemo kot lemo:

Lema 3.1 ([1]) *Skupna amortizirana zahtevnost in skupna dejanska zahtevnost zaporedja n operacij sta povezani na naslednji način:*

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

Če potencialno funkcijo Φ definiramo tako, da je $\Phi(D_n) \geq \Phi(D_0)$, potem je $\Phi(D_n) - \Phi(D_0) \geq 0$. Sedaj lahko enačbo v lemi 3.1 omejimo na naslednji način:

$$\begin{aligned} \sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\ &\geq \sum_{i=1}^{n} c_i. \end{aligned} \tag{3.3}$$

Iz neenačbe 3.3 sledi, da je amortizirana zahtevnost $\sum_{i=1}^{n} \widehat{c_i}$ zgornja meja skupne dejanske zahtevnosti $\sum_{i=1}^{n} c_i$.

V praksi običajno ne vemo, koliko operacij bomo izvedli, zato zahtevamo, da za potencial podatkovne strukture D_i, ki jo dobimo po izvedbi i-te operacije na podatkovni strukturi D_{i-1}, velja $\Phi(D_i) \geq \Phi(D_0)$ za vsak i, in s tem zagotovimo, kot pri računovodski metodi, da plačamo stroške operacije v naprej. Običajno definiramo, da je $\Phi(D_0) = 0$ in potem dokažemo, da je $\Phi(D_i) \geq 0$ za vsak i [1, 4].

Zgled. Zgled ponovno izvedemo na podatkovni strukturi sklad, na katerem izvajamo operacije *push*, *pop* in *multipop*. Definirajmo potencialno funkcijo Φ na skladu kot število elementov v skladu. Začnemo s praznim skladom D_0, za katerega velja, da je $D_0 = 0$. Ker velikost sklada ni nikoli negativno število, ima sklad D_i, ki ga dobimo po i-ti operaciji, nenegativen potencial, zato je

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0). \end{aligned} \tag{3.4}$$

Skupna amortizirana zahtevnost n operacij glede na Φ tako predstavlja zgornjo mejo dejanske zahtevnosti.

Spomnimo. V zgledih iz razdelkov 3.1 in 3.2 smo ugotovili, da imajo operacije *push*, *pop* in *multipop* dejanske časovne zahtevnosti, kot je zapisano v tabeli 3.2.

Operacija	Dejasnka časovna zahtevnost c_i
push	1
pop	1
multipop	$\min\{\|S\|, k\}$

Tabela 3.2: V tabeli so zapisane dejanske časovne zahtevnosti na skladu.

Sedaj bomo z uporabo metode potencialov izračunali amortizirane zahtevnosti operacij na skladu. Če je i-ta operacija sklada velikosti s operacija *push*, potem je potencialna razlika

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

Ker je dejanska časovna zahtevnost c_i za operacijo *push* enaka 1, potem po enačbi 3.1 sledi, da je amortizirana zahtevnost operacije *push* enaka

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

Recimo, da je i-ta operacija na skladu velikosti s operacija $multipop(S, k)$, ki odstrani $k' = \min\{s, k\}$ zadnjih elementov s sklada. Iz tabele 3.2 vidimo, da je dejanska časovna zahtevnost c_i operacije *multipop* enaka $k' = \min\{s, k\}$, zato je potencialna razlika enaka

$$\Phi(D_i) - \Phi(D_{i-1}) = s - k' - s = -k'.$$

Tako je amortizirana zahtevnost operacije *multipop* enaka

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$

Podobno izračunamo, da je amortizirana zahtevnost operacije *pop* enaka 0.

Amortizirana zahtevnost vsake od treh omenjenih operacij je $O(1)$, zato je skupna amortizirana zahtevnost zaporedja n operacij enaka $O(n)$. Na začetku zgleda smo že ugotovili, da je $\Phi(D_i) \geq \Phi(D_{i-1})$, zato je skupna amortizirana zahtevnost n operacij zgornja meja skupne dejanske zahtevnosti. Tako dobimo, da je zahtevnost zaporedja n operacij v najslabšem primeru enaka $O(n)$.

Poglavje 4

Amortizirana zahtevnost lomljenih dreves

Najprej definirajmo nekaj oznak. Naj bo T lomljeno drevo, na katerem izvedemo operacijo vstavljanja ali iskanja. Naj T_i označuje drevo, ki se je po i-tem koraku lomljenja spremenilo in $T_0 = T$. Če je x poljubno vozlišče v drevesu T_i, potem s $T_i(x)$ označimo poddrevo T_i s korenom v x in z $|T_i(x)|$ označimo število vozlišč tega poddrevesa.

Predpostavimo, da vozlišče x ni koren drevesa T. Uporabimo operacijo lomljenje od spodaj navzgor nad vozliščem x. Recimo, da je vozlišče x po m korakih lomljenja (t.j. izvedba m-tih rotacij CIK, CAK, CIK-CIK, CAK-CAK, CIK-CAK, CAK-CIK) koren drevesa.

Definicija 4.1 ([5]) *Za vsak korak lomljenja i in vsako vozlišče x v drevesu T, definiramo rang vozlišča x na koraku i kot*

$$r_i(x) = \log |T_i(x)|.$$

Če je vozlišče x list drevesa, potem je $|T_i(x)| = 1$, zato je $r_i(x) = 0$. Vozlišča, ki so bolj na dnu drevesa imajo nižji rang, medtem ko ima koren najvišji rang v drevesu.

Koliko dela mora algoritem opraviti, da poišče ali vstavi vozlišče v poddrevo, je povezano z višino poddrevesa in s tem tudi z rangom korena tega poddrevesa. Zato bi radi definirali potencialno funkcijo na takšen način, da bi višja drevesa imela večji potencial in nižja drevesa bi imela manjši potencial, saj se količina dela poveča, če je drevo višje. Pri tem si pomagamo s funkcijo *rang*.

Definicija 4.2 ([5]) *Naj T_i označuje drevo T po i-tem koraku lomljenja in $T_0 = T$. Potencialna funkcija drevesa T_i je definirana kot vsota rangov vseh vozlišč drevesa na koraku*

i:

$$\Phi(T_i) = \sum_{x \in T_i} r_i(x).$$

Če je drevo prazno ali pa vsebuje samo eno vozlišče, potem je potencialna funkcija drevesa enaka 0. Z večanjem drevesa se viša tudi potencialna funkcija.

Amortizirano zahtevnost $\widehat{c_i}$ koraka i dobimo z naslednjo enačbo

$$\widehat{c_i} = c_i + \Phi(T_i) - \Phi(T_{i-1}),$$

kjer je c_i dejanska zahtevnost koraka i in $\Phi(T_i) - \Phi(T_{i-1})$ predstavlja razliko potencialov dreves T_i in T_{i-1}, kjer drevo T_i dobimo iz drevesa T_{i-1} z uporabo i-te rotacije (t.j. CIK, CIK-CIK, CIK-CAK, ...).

Naš glavni cilj je, da z uporabo definicij in načina delovanja operacije lomljenja ugotovimo meje za $\widehat{c_i}$, kar bo pomagalo pri ugotavljanju amortizirane zahtevnosti celotnega procesa lomljenja (iskanje in vstavljanje). Najprej potrebujemo pomožno trditev.

Lema 4.3 ([9]) *Če so α, β in γ pozitivna realna števila z lastnostjo $\alpha + \beta \leq \gamma$, potem je*

$$\log_2 \alpha + \log_2 \beta \leq 2 \log_2 \gamma - 2.$$

Dokaz. Očitno velja, da je $(\sqrt{\alpha} - \sqrt{\beta})^2 \geq 0$, saj je kvadrat poljubnega realnega števila nenegativen. To lahko poenostavimo kot

$$\begin{aligned} \alpha - 2\sqrt{\alpha\beta} + \beta &\geq 0 \\ 2\sqrt{\alpha\beta} &\leq \alpha + \beta \\ \sqrt{\alpha\beta} &\leq \frac{\alpha + \beta}{2}. \end{aligned}$$

Ker je $\alpha + \beta \leq \gamma$, sledi, da je $\sqrt{\alpha\beta} \leq \frac{\gamma}{2}$. Če na to enačbo delujemo z logaritmom z osnovo 2, dobimo rezultat leme.

$$\begin{aligned} \log_2(\sqrt{\alpha + \beta}) &\leq \log_2 \frac{\gamma}{2} \\ \frac{1}{2}\log_2 \alpha + \frac{1}{2}\log_2 \beta &\leq \log_2 \gamma - \log_2 2 \\ \frac{1}{2}\log_2 \alpha + \frac{1}{2}\log_2 \beta &\leq \log_2 \gamma - 1 \\ \log_2 \alpha + \log_2 \beta &\leq 2\log_2 \gamma - 2. \end{aligned}$$

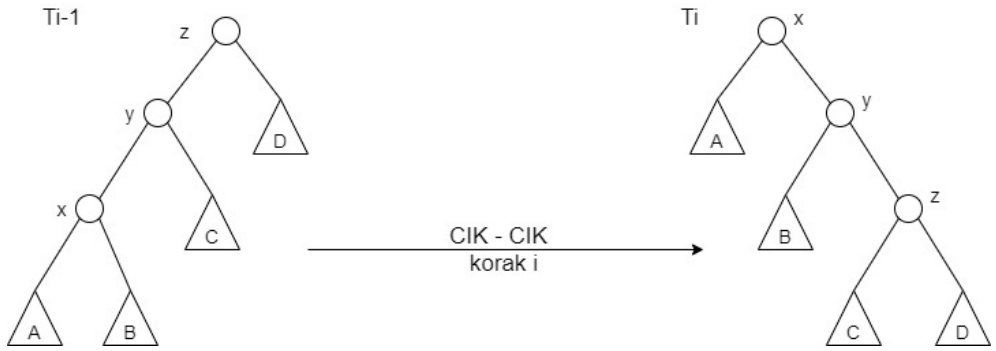
□

Nadalje bomo analizirali različne možnosti lomljenja posebej.

Lema 4.4 ([5]) *Če je i-ti korak lomljenja pri vozlišču x korak CIK-CIK ali CAK-CAK, potem amortizirana zahtevnost $\widehat{c_i}$ zadošča neenačbi*

$$\widehat{c_i} < 3(r_i(x) - r_{i-1}(x)).$$

Dokaz. Imamo naslednji primer:



Slika 4.1: i-ti korak lomljenja je korak CIK-CIK.

Dejanska zahtevnost c_i koraka CIK-CIK ali CAK-CAK je 2 enoti in se v tem koraku spremenijo samo velikosti poddreves s koreni v vozliščih x, y in z. Zato dobimo naslednjo enačbo:

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 2 + r_i(x) + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) - r_{i-1}(z) \\ &= 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y)\end{aligned} \tag{4.1}$$

Zadnjo vrstico dobimo z logaritmom od $|T_i(x)| = |T_{i-1}(z)|$ (oz. $r_i(x) = r_{i-1}(z)$), kar izhaja iz ugotovitve, da ima poddrevo s korenom z pred lomljenjem enako velikost kot poddrevo s korenom x po lomljenju. Sedaj uporabimo lemo 4.3, kjer iz enačbe izničimo 2 (dejansko zahtevnost). Naj bo $\alpha = |T_{i-1}(x)|$, $\beta = |T_i(z)|$ in $\gamma = |T_i(x)|$. Iz slike 4.1 vidimo, da $T_{i-1}(x)$ vsebuje x ter poddrevesi A in B; $T_i(z)$ vsebuje z ter poddrevesi C in D; $T_i(x)$ vsebuje vse že naštete komponente (in tudi y). Ker je $\alpha + \beta < \gamma$, lahko uporabimo lemo 4.3 in dobimo

$$r_{i-1}(x) + r_i(z) \leq 2r_i(x) - 2 \text{ oz.}$$

$$2r_i(x) - r_{i-1}(x) - r_i(z) - 2 \geq 0.$$

Če dobljeno neenakost uporabimo v enačbi 4.1 za $\widehat{c_i}$, dobimo

$$\begin{aligned}\widehat{c_i} &\leq 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) + 2r_i(x) - r_{i-1}(x) - r_i(z) - 2 \\ &= 2r_i(x) - 2r_{i-1}(x) + r_i(y) - r_{i-1}(y) \\ &= 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y)\end{aligned} \tag{4.2}$$

Pred korakom i je y starš od x, zato je $|T_{i-1}(y)| > |T_{i-1}(x)|$. Po koraku i je x starš od y in zato je $|T_i(x)| > |T_i(y)|$. Če vzamemo logaritme, dobimo $r_{i-1}(y) > r_{i-1}(x)$ in $r_i(x) < r_i(y)$. Tako na koncu dobimo

$$\widehat{c_i} < 3r_i(x) - 3r_{i-1}(x).$$

□

Lema 4.5 ([5]) *Če je i-ti korak lomljenja pri vozlišču x korak CIK-CAK ali CAK-CIK, potem amortizirana zahtevnost $\widehat{c_i}$ zadošča neenačbi*

$$\widehat{c_i} < 2(r_i(x) - r_{i-1}(x)).$$

Lema 4.6 ([5]) *Če je i-ti korak lomljenja pri vozlišču x korak CIK ali CAK, potem amortizirana zahtevnost $\widehat{c_i}$ zadošča neenačbi*

$$\widehat{c_i} < 1 + (r_i(x) - r_{i-1}(x)).$$

Dokaz leme 4.5 je podoben kot dokaz leme 4.4 in dokaz leme 4.6 je očiten.

Za skupno amortizirano zahtevnost operacij iskanja in vstavljanja moramo dodati zahtevnost vseh korakov lomljenja, ki se izvedejo med iskanjem ali vstavljanjem. Če je korakov m, potem je lahko največ en korak (zadnji) CIK ali CAK korak, za katerega velja lema 4.6; vsi ostali koraki pa izpolnjujejo meje v lemah 4.4 in 4.5 [5]. Tako dobimo, da je skupna amortizirana zahtevnost enaka

$$\begin{aligned}\sum_{i=1}^{m} \widehat{c_i} &= \sum_{i=1}^{m-1} \widehat{c_i} + \widehat{c_m} \\ &\leq \sum_{i=1}^{m-1} (3r_i(x) - 3r_{i-1}(x)) + (1 + 3r_m(x) - 3r_{m-1}(x)) \\ &= 1 + 3r_m(x) - 3r_0(x).\end{aligned} \tag{4.3}$$

Z uporabo prejšnjih lem in izpeljave 4.3, smo dokazali lemo, ki ji pravimo *lema dostopa*.

Lema 4.7 (Lema dostopa [9]) *Naj bo x iskano vozlišče. Če uporabimo operacijo lomljenje na vozlišču x, potem je amortizirana zahtevnost lomljenja največ*

$$3(r(koren) - r(x)) + 1.$$

Če izpeljavo 4.3 še nadalje ocenimo, dobimo, da je

$$\begin{aligned} \sum_{i=1}^{m} \widehat{c}_i &\leq 1 + 3r_m(x) - 3r_0(x) \\ &\leq 1 + 3r_m(x) \\ &= 1 + 3\log n. \end{aligned} \tag{4.4}$$

Ker je $r_0(x) \geq 0$, lahko ocenimo, da je desna stran brez začetnega ranga večja in ker je na koncu lomljenja vozlišče x koren drevesa, dobimo, da je $r_m(x) = \log n$, kjer je n število vozlišč drevesa [5].

S tem smo dokazali naslednji izrek:

Izrek 4.8 ([5]) *Amortizirana zahtevnost operacij iskanja ali vstavljanja z lomljenjem v lomljenih drevesih z n vozlišči je navzgor omejena z*

$$1 + 3\log n,$$

kjer iskano vozlišče premikamo navzgor po drevesu.

4.1 Teoretična amortizirana zahtevnost lomljenih dreves

Amortizirano analizo lomljenih dreves lahko posplošimo na naslednji način [9]:

- vsakemu vozlišču x dodelimo pozitivno utež $w(x)$,
- z $s(x)$ definiramo vsoto uteži vseh vozlišč v poddrevesu s korenom x (vključno z x),
- rang vozlišča x definiramo kot $r(x) = log(s(x))$,
- potencial drevesa Φ definiramo kot vsoto rangov vseh vozlišč drevesa in

- kot merilo časa delovanja lomljenja uporabimo število opravljenih operacij, razen če ni bilo ponovitev, v tem primeru izračunamo samo enega.

Recimo, da imamo zaporedje m operacij iskanja na lomljenem drevesu z n vozlišči. Če uteži vseh ključev ostanejo nespremenjene, je razlika med začetnim in končnim potencialom nad zaporedjem največ

$$\sum_{i=1}^{n} \log\left(\frac{W}{w(i)}\right),$$

kjer je $W = \sum_{i=1}^{n} w(i)$, saj je velikost vozlišča, ki vsebuje ključ i, največ W in najmanj $w(i)$ [9].

Izrek 4.9 (Izrek ravnotežja [9]) *Naj bo n število vozlišč lomljenega drevesa in naj bo m število operacij iskanja. Skupni čas operacije iskanja je*

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

Dokaz. Vsakem ključu dodelimo utež $w(i) = \frac{1}{n}$, za vsak $i \in \{1, \ldots, n\}$. Potem je

$$W = \sum_{i=1}^{n} \frac{1}{n} = \frac{1}{n} \sum_{i=1}^{n} 1 = \frac{n}{n} = 1.$$

Ker je $W = 1$, sledi, da je $r(koren) = 0$. Po lemi 4.7 je amortizirani čas operacije iskanja največ

$$3(r(koren) - r(i)) + 1 = 3\left(0 - \log\left(\frac{1}{n}\right)\right) + 1 = 3\log(n) + 1.$$

Če izvedemo m operacij iskanja dobimo, da je amortizirana zahtevnost največ $O(m + m\log(n))$.

Razlika med začetnim in končnim potencialom nad zaporedjem je največ

$$\sum_{i=1}^{n} \log\left(\frac{W}{w(i)}\right) = \sum_{i=1}^{n} \log\frac{1}{\frac{1}{n}} = \log(n) \sum_{i=1}^{n} 1 = n\log(n).$$

Sledi, da je skupna zahtevnost operacije iskanja enaka

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

□

Za vsak ključ i naj bo $q(i)$ število operacij iskanja ključa i.

Izrek 4.10 (Izrek statične optimalnosti [9]) *Naj bo n število vozlišč lomljenega drevesa in naj bo m število operacij iskanja. Če vsak ključ iščemo vsaj enkrat, potem je skupni čas operacije iskanja*

$$O\left(m + \sum_{i=1}^{n} q(i) \log\left(\frac{m}{q(i)}\right)\right).$$

Dokaz. Naj ima ključ i utež $w(i) = \frac{q(i)}{m}$. Potem je

$$W = \sum_{i=1}^{n} w(i) = \sum_{i=1}^{n} \frac{q(i)}{m} = \frac{1}{m} \sum_{i=1}^{n} q(i) = 1.$$

Ker je $W = 1$, sledi, da je $r(koren) = 0$. Po lemi 4.7 je amortizirani čas iskanja ključa i največ

$$3(r(koren) - r(i)) + 1 = 3\left(0 - \log\left(\frac{q(i)}{m}\right)\right) + 1 =$$

$$= 3 \log\left(\frac{m}{q(i)}\right) + 1.$$

Če izvedemo m operacij iskanja, dobimo, da je amortizirana zahtevnost največ

$$O\left(m + \sum_{i=1}^{n} q(i) \log\left(\frac{m}{q(i)}\right)\right).$$

Razlika med začetnim in končnim potencialom nad zaporedjem je največ

$$\sum_{i=1}^{n} \log\left(\frac{W}{w(i)}\right) = \sum_{i=1}^{n} \log\left(\frac{1}{\frac{q(i)}{m}}\right) = \sum_{i=1}^{n} \log\left(\frac{m}{q(i)}\right).$$

Sledi, da je skupna zahtevnost operacije iskanja enaka

$$O\left(m + \sum_{i=1}^{n} q(i) \log\left(\frac{m}{q(i)}\right)\right).$$

□

Recimo, da so ključi oštevilčeni od 1 do n v naraščajočem vrstnem redu. Naj bo $i_1, i_2, \ldots, i_m$ zaporedje iskanih ključev.

Izrek 4.11 (Izrek fiksnega ključa [9]) *Naj bo n število vozlišč lomljenega drevesa in naj bo m število operacij iskanja. Če je f poljubni ključ lomljenega drevesa, potem je skupni*

čas operacije iskanja

$$O\left(n\log(n)+m+\sum_{j=1}^{m}\log(\mid i_j-f\mid+1)\right).$$

Dokaz. Naj ima ključ i utež $w(i)=\frac{1}{(|i-f|+1)^2}$. Potem je

$$W=\sum_{i=1}^{n}w(i)=\sum_{i=1}^{n}\frac{1}{(\mid i-f\mid+1)^2}=O(1).$$

Ker je $W=1$, sledi, da je $r(koren)=0$. Po lemi 4.7 je amortizirani čas j-te operacije iskanja največ

$$3(r(koren)-r(j))+1=3\left(0-\log\left(\frac{1}{(\mid i_j-f\mid+1)^2}\right)\right)=$$

$$=3\log(\mid i_j-f\mid+1)^2+1=6\log(\mid i_j-f\mid+1)+1.$$

Če izvedemo m operacij iskanja, dobimo, da je amortizirana zahtevnost največ

$$O\left(m+\sum_{j=1}^{m}\log(\mid i_j-f\mid+1)\right).$$

Razlika med začetnim in končnim potencialom nad zaporedjem je

$$\sum_{i=1}^{n}\log\left(\frac{W}{w(i)}\right)\leq\sum_{i=1}^{n}\log\left(\frac{1}{\frac{1}{n^2}}\right)=\sum_{i=1}^{n}\log(n^2)=2nlog(n)=O(n\log(n)),$$

saj je utež poljubnega ključa vsaj $\frac{1}{n^2}$. Sledi, da je skupna zahtevnost operacije iskanja enaka

$$O\left(n\log(n)+m+\sum_{j=1}^{m}\log(\mid i_j-f\mid+1)\right).$$

□

Izrek 4.12 (Izrek delovnega niza [9]) *Naj bo n število vozlišč lomljenega drevesa in naj bo m število operacij iskanja. Naj a_j predstavlja ključ, ki ga iščemo v j-tem koraku. Za poljuben korak j definiramo $t(j)$ kot število različnih ključev, ki smo jih iskali med zadnjim iskanjem ključa a_j in korakom j oz. od začetka operacije iskanja, če ključ a_j prvič iščemo.*

Skupni čas operacije iskanja je

$$O\left(n\log(n)+m+\sum_{j=1}^{m}\log(t(j)+1)\right).$$

Dokaz. Ključem določimo uteži $1, \frac{1}{4}, \frac{1}{9}, \ldots, \frac{1}{n^2}$ po vrstnem redu glede na iskanje (ključ, katerega iščemo najprej, dobi največjo utež, vsi ključi, katerih nismo iskali, dobijo najmanjše uteži). Po vsakem iskanju ponovno določimo uteži na naslednji način: recimo, da ima ključ a_j med j-tim iskanjem utež enako $\frac{1}{k^2}$. Po j-tem iskanju dobi ključ a_j utež 1 in vsak ključ a_i z utežjo $\frac{1}{(k')^2}$, kjer $k' < k$, dobi utež $\frac{1}{(k'+1)^2}$. Ta porazdelitev permutira uteži $1, \frac{1}{4}, \frac{1}{9}, \ldots, \frac{1}{n^2}$ med ključi. Poleg tega zagotavlja, da bo med j-tim iskanjem ključ a_j imel utež enako $\frac{1}{(t(j)+1)^2}$. Tako dobimo, da je

$$W=\sum_{k=1}^{n}w(k)=\sum_{k=1}^{n}\frac{1}{k^2}\leq\sum_{k=1}^{\infty}\frac{1}{k^2}=\frac{\pi^2}{6}=O(1).$$

Po lemi 4.7 je amortizirani čas j-te operacije iskanja največ

$$3(r(koren)-r(a_j))+1\leq 3\left(\log\frac{\pi^2}{6}-\log\left(\frac{1}{(t(j)+1)^2}\right)\right)+1$$

$$=3\left(\log\frac{\pi^2}{6}+2\log(t(j)+1)\right)+1.$$

Če izvedemo m operacij iskanja, dobimo, da je amortizirana zahtevnost največ

$$O\left(m+\sum_{j=1}^{m}\log(t(j)+1)\right).$$

Razlika med začetnim in končnim potencialom nad zaporedjem je

$$\sum_{i=1}^{n}\log\left(\frac{W}{w(a_j)}\right)\leq\sum_{i=1}^{n}\log\left(\frac{\frac{\pi^2}{6}}{\frac{1}{n^2}}\right)=\sum_{i=1}^{n}\log\left(\frac{\pi^2n^2}{6}\right)$$

$$=2\log\left(\frac{n\pi}{6}\right)\sum_{i=1}^{n}1=2n\log\left(\frac{n\pi}{6}\right)$$

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

Sledi, da je skupna zahtevnost operacije iskanja enaka

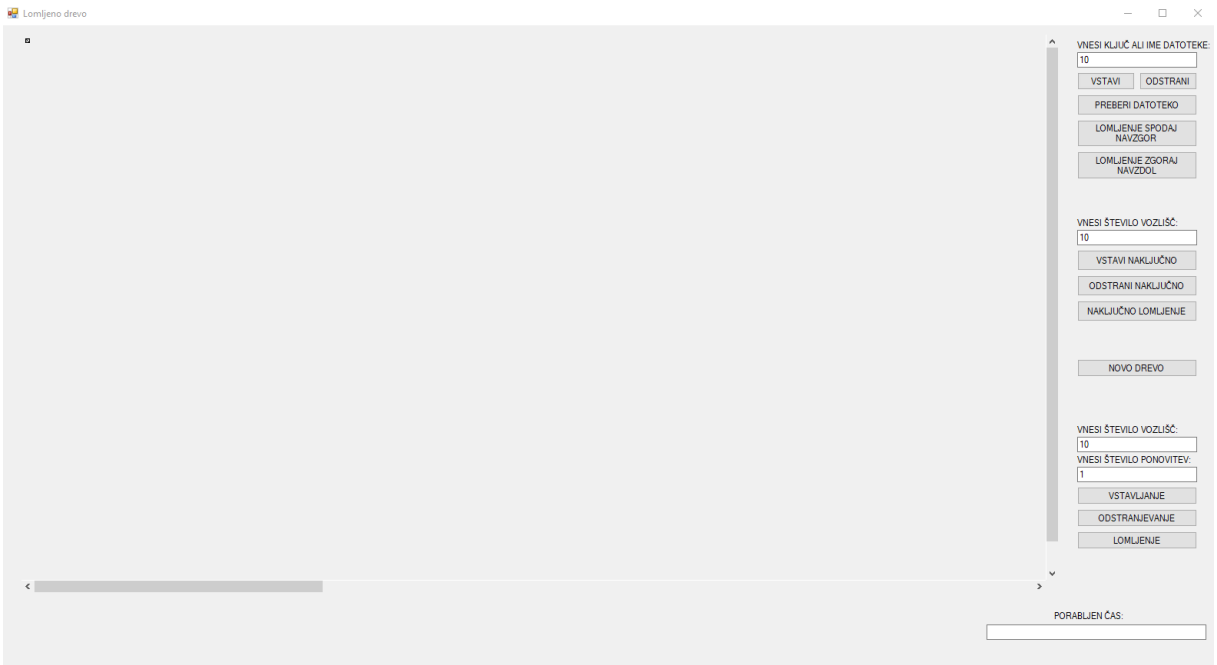
$$O\left(n\log(n)+m+\sum_{j=1}^{m}\log(t(j)+1)\right).$$

□

Poglavje 5

Implementacija

Podatkovno strukturo lomljena drevesa smo implementirali s programom Microsoft Visual Studio v programskem jeziku C++. Za lažjo preglednost, analizo in predstavitev so lomljena drevesa implementirana tudi grafično. Na sliki 5.1 je prikazano okno, ki se odpre ob zagonu programa.

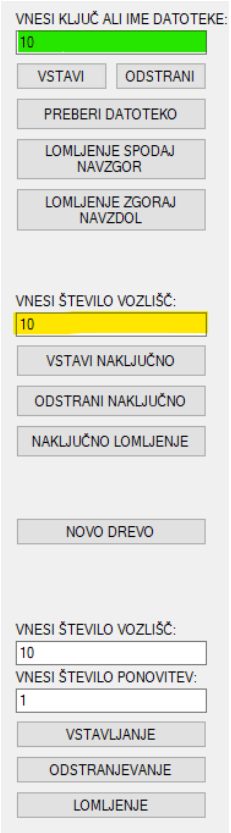


Slika 5.1: Ob zagonu programa, se odpre novo okno.

5.1 Vhodni podatki

Ob zagonu programa je lomljeno drevo prazno. Imamo tri možnosti za kreiranje drevesa:

1. V vnosno polje (na sliki 5.2 označeno z zeleno barvo) zapišemo vrednost ključa, ki ga želimo vstaviti v drevo in izvedemo operacijo *vstavi* (glej 5.2.4) s klikom na gumb VSTAVI. Ključ, ki ga na novo vstavimo, bo vedno koren drevesa.



Slika 5.2: Pri vstavljanju naključnih vrednosti v drevo, zapišemo vrednost v ustrezno vnosno polje.

2. V vnosno polje (na sliki 5.2 označeno z zeleno barvo) zapišemo ime tekstovne datoteke. Za imenom datoteke moramo obvezno zapisati tudi končnico *.txt* (npr. *primer.txt*), sicer program ne naredi ničesar. Na koncu pritisnemo gumb PREBERI DATOTEKO (glej razdelek 5.2.1). Program to datoteko prebere in ustvari drevo z zapisanimi ključi. Datoteka mora biti naslednje oblike. V prvi vrstici je število vozlišč drevesa, v naslednji vrstici pa so našteti ključi drevesa.

3. Drevo lahko ustvarimo tudi z naključno izbranimi števili (glej tudi 5.3.1). To naredimo tako, da v vnosno polje (na sliki 5.2 označeno z rumeno barvo) zapišemo celo število

(recimo, da je to število n), ki predstavlja število vozlišč novega drevesa, in pritisnemo gumb VSTAVI NAKLJUČNO. Program naključno vstavi ključe od 1 do n v drevo.

5.2 Delovanje programa

V programski kodi je implementiran razred *Vozlisce*, ki predstavlja posamezno vozlišče drevesa. Lastnosti razreda so celo število, ki ustreza ključu vozlišča, ter kazalci na starša, levega in desnega otroka vozlišča. Definirana sta tudi dva konstrukturja s katerima ustvarimo novo vozlišče. 5.1 predstavlja strukturo razreda *Vozlisce*.

```
class Vozlisce{
    public:
        Vozlisce (int aK, Vozlisce* aL, Vozlisce* aD, Vozlisce* aS);
        Vozlisce (int aK);
        \sim Vozlisce ();

        int kljuc;
        Vozlisce *levi, *desni, *stars;

    protected:
    private:
}
```

Algoritem 5.1: Struktura razreda *Vozlisce*.

Za samo delovanje programa je implementiran razred *DIDrevo*, ki predstavlja lomljeno drevo. V samem razredu imamo spremenljivko *koren*, ki kaže na koren lomljenega drevesa. V nadaljevanju so predstavljene metode, ki so definirane znotraj razreda *DIDrevo*, za delovanje lomljenih dreves.

5.2.1 Metoda *preberiDrevo*

Metoda *preberiDrevo* najprej preveri, ali je trenutno drevo prazno. Če drevo ni prazno, potem metoda *novoDrevo* iz trenutnega drevesa odstrani vsa vozlišča. Nato iz datoteke najprej preberemo število vozlišč novega drevesa in potem izvedemo zanko, kjer preberemo ključ iz datoteke in ga vstavimo v drevo. Ko datoteko preberemo do konca, jo zapremo. 5.2 predstavlja psevdokodo metode *preberiDrevo*.

```
void preberiDrevo(string datoteka){
    ifstream dat;
    dat.open(datoteka.c_str());
    if (dat.good()){
        //ce drevo ni prazno, potem odstranimo vsa trenutna vozlisca
        if (!jePrazno())
            novoDrevo();
        int n, aK;
        dat >> n; //preberemo stevilo vozlisc
        //preberemo kljuce in jih vsatvimo v drevo
        for (int i = 0; i < n; i++){
            dat >> aK;
            vstaviNovoDrevo(aK);
            if(dat.fail())
                break;
        }
        dat.close(); //datoteko zapremo
    }
}
```

Algoritem 5.2: Metoda *preberiDrevo* prebere datoteko in vstavi ključe v drevo.

5.2.2 Operacija lomljenje od spodaj navzgor

Operacija lomljenje od spodaj navzgor je v programu definirana z metodo *lomljenje*1, ki kot parameter prejme vozlišče aV, ki je po končani operaciji v korenu drevesa. Postopek lomljenja od spodaj navzgor je podrobneje opisan v razdelku 2.1.1.

Metoda *lomljenje*1 je zapisana v psevdokodi 5.3.

```
void lomljenje1(Vozlisce* aV){
    if(jePrazno())
        return;
    if(koren->levi == nullptr && koren->desni == nullptr)
        return;
    Vozlisce* s;
    Vozlisce* e;
```

```
    //naslednji postopek izvajamo dokler aV ni koren drevesa
    while(aV->stars != nullptr){
        s = aV->stars;
        e = s->stars;
        if(aV == s->levi){ //aV je levi otrok
            if(s->stars == nullptr) //CIK
                zavrtiDesno(s);
            else if(e->levi == s){ //CIK-CIK
                zavrtiDesno(e);
                zavrtiDesno(s);
            }
            else if(e->desni == s){ //CAK-CIK
                zavrtiDesno(s);
                zavrtiLevo(e);
            }
        }
        else if(aV == s->desni){ //aV je desni otrok
            if(s->stars == nullptr) //CAK
                zavrtiLevo(s);
            else if(e->desni == s){ //CAK-CAK
                zavrtiLevo(e);
                zavrtiLevo(s);
            }
            else if(e->levi == s){ //CIK-CAK
                zavrtiLevo(s);
                zavrtiDesno(e);
            }
        }
    }
}
```

Algoritem 5.3: Metoda *lomljenje*1 s katero izvedemo lomljenje od spodaj navzgor nad vozliščem aV.

Opomba 5.1 *Metoda jePrazno() v psevdokodi 5.3 preveri, ali je drevo prazno. Drevo je prazno, če koren drevesa ni definiran oz. če je koren == nullptr.*

Opomba 5.2 *V primeru, da ključa, nad katerim izvedemo lomljenje od spodaj navzgor, ni*

v drevesu, se lomljenje izvede nad zadnjim neničelnem vozlišču, kjer bi pričakovali, da leži ta ključ.

Opomba 5.3 *Metoda* zavrtiDesno *izvede enojno rotacijo (glej 2.1.1). Metoda* zavrtiLevo *je simetrična metodi* zavrtiDesno.

5.2.3 Operacija lomljenje od zgoraj navzdol

Operacija lomljenje od zgoraj navzdol je v programu definirana z metodo *lomljenje*2, ki kot parameter prejme celo število aK oz. ključ, nad katerim želimo izvesti operacijo. Torej aK predstavlja ključ vozlišča, ki je po končani operaciji v korenu drevesa. Postopek lomljenja od zgoraj navzdol je podrobneje opisan v razdelku 2.1.2.

Metoda *lomljenje*2 je zapisana v psevdokodi 5.4.

```
void lomljenje2(int aK){
    Vozlisce* trenutni = koren; //kaze na koren osrednjega poddrevesa
    Vozlisce* otrok; //kaze na levega ali desnega otroka od trenutni,
                     //odvisno katerega potrebujemo
    Vozlisce* zadnjiMali = nullptr; //kaze na vozlisce z najvecjim
                                    // kljucem poddrevesa z manjsimi kljuci
    Vozlisce* prviVecji = nullptr; //kaze na vozlisce z najmanjsim
                                   // kljucem poddrevesa z vecjimi kljuci

    //naslednji postopek izvajamo dokler se imamo kaksno vozlisce
    //v osrednjem poddrevesu in aK ni kljuc korena tega poddrevesa
    while(trenutni != nullptr && trenutni->kljuc != aK){
        if(aK < trenutni->kljuc){
            otrok = trenutni->levi;
            if(otrok == nullptr || aK == otrok->kljuc) //CIK
                poveziDesno(trenutni, prviVecji);
            else if(aK < otrok->kljuc){ //CIK-CIK
                rotacijaDesno(trenutni);
                otrok = trenutni->levi;
                poveziDesno(trenutni, prviVecji);
            }
            else{ //CIK-CAK
                poveziDesno(trenutni, prviVecji);
```

```
                povezi Levo(trenutni, zadnjiMali);
            }
        }
        else{ //aK < trenutni->kljuc
            otrok = trenutni->desni;
            if(otrok == nullptr || aK == otrok->kljuc) //CAK
                povezi Levo(trenutni, zadnjiMali);
            else if(aK > otrok->kljuc){ //CAK-CAK
                rotacijaLevo(trenutni);
                otrok = trenutni->desni;
                povezi Levo(trenutni, zadnjiMali);
            }
            else{ //CAK-CIK
                povezi Levo(trenutni, zadnjiMali);
                povezi Desno(trenutni, prviVecji);
            }
        }
    }
    //ce kljuca aK nismo nasli, ustavrimo novo vozlisce s kljucem aK
    if(trenutni == nullptr){
        trenutni = new Vozlisce(aK, nullptr, nullptr)
        //poddrevesa ustrezno spojimo skupaj
        if(zadnjiMali != nullptr){
            zadnjiMali->desni = nullptr;
            trenutni->levi = poisciStarsa(zadnjiMali);
            trenutni->levi->stars = trenutni;
        }
        if(prviVecji != nullptr){
            prviVecji->levi = nullptr;
            trenutni->desni = poisciStarsa(prviVecji);
            trenutni->desni->stars = trenutni;
        }
    }
    //ce smo kljuc aK nasli
    else{
        if(zadnjiMali != nullptr){
            zadnjiMali->desni = trenutni->levi;
            if(zadnjiMali->desni != nullptr)
```

```
                zadnjiMali->desni->stars = zadnjiMali;
            trenutni->levi = poisciStarsa(zadnjiMali);
            trenutni->levi->stars = trenutni;
        }
        if(prviVecji != nullptr){
            prviVecji->desni = trenutni->desni;
            if(prviVecji->levi != nullptr)
                prviVecji->levi->stars = prviVecji;
            trenutni->desni = poisciStarsa(prviVecji);
            trenutni->desni->stars = trenutni;
        }
    }
    koren = trenutni;
}
```

Algoritem 5.4: Metoda *lomljenje*2 s katero izvedemo lomljenje od zgoraj navzdol nad vozlišču s ključem aK.

Opomba 5.4 *Metoda* poveziLevo *je simetrična metodi* poveziDesno *(glej razdelek 2.1.2).*

```
void poveziDesno(Vozlisce* &trenutni, Vozlisce* &prviVecji){
    if(prviVecji == nullptr){
        prviVecji = trenutni;
        trenutni = trenutni->levi;
        if(trenutni != nullptr)
            trenutni->stars = nullptr;
        prviVecji = nullptr;
    }
    else{
        prviVecji = trenutni;
        trenutni->stars = prviVecji;
        prviVecji = prviVecji->levi;
        trentuni = trenutni->levi;
        if(trenutni != nullptr)
            trenutni->stars = nullptr;
        prviVecji->levi = nullptr;
    }
```

```
}
```

Algoritem 5.5: Metoda *poveziDesno* preko parametra prejme kazalca *trenutni* in *prviVecji*.

Opomba 5.5 *Metoda* rotacijaDesno *(glej psevdokodo 5.6) je podobna metodi* zavrtiDesno. *Razlika med njima je, da pri metodi* rotacijaDesno *vedno izvedemo rotacijo korena drevesa. Razlika je tudi v tem, da kazalec* trenutni *prenašamo po referenci, torej po izvedbi metode* rotacijaDesno *kazalec* trenutni *kaže na novo vozlišče. Metoda* rotacijaLevo *je simetrična metodi* rotacijaDesno.

```
void poveziDesno(Vozlisce* &trenutni){
    Vozlisce* o = trenutni->levi;
    Vozlisce* e;
    //ustrezno spremenimo kazalce
    trenutni->levi = o->desni;
    o->desni = trenutni;
    o->stars = nullptr;
    trenutni->stars = o;
    //trenutni postane novi koren osrednjega poddrevesa
    trenutni = o;
}
```

Algoritem 5.6: Metoda *rotacijaDesno* preko parametra prejme kazalec *trenutni*.

5.2.4 Operacija vstavljanja ključa v drevo

Za vstavljanje poljubnega ključa v lomljeno drevo je definirana metoda *vstavi*, ki preko parametra prejme celo število aK, ki ga nato vstavi v drevo. Postopek vstavljanja vozlišča v lomljeno drevo je podrobneje opisan v razdelku 2.2.2.

Opomba 5.6 *Če je vozlišče* aK *že v drevesu, potem samo izvedemo metodo* lomljenje1 *nad vozliščem* p.

Metoda *vstavi* je zapisana v psevdokodi 5.7.

```
void vstavi(int aK){
    if(jePrazno())
        koren = new Vozlisce(aK, nullptr, nullptr, nullptr);
    else{
        Vozlisce* p = koren; //s p se premikamo po drevesu navzdol
        Vozlisce* e = nullptr; //predstavljal bo list drevesa
        while(p != nullptr){
            e = p;
            //ce je aK v drevesu, potem funkcijo zakljucimo
            if(aK == p->kljuc){
                lomljenje1(p);
                return;
            }
            if(aK < p->kljuc)
                p = p->levi;
            else
                p = p->desni;
        }
        //novo vozlisce bo otrok od e in
        //lomimo na novo vstavljenem vozliscu
        if(aK < e->kljuc){
            e->levi = new Vozlisce(aK, nullptr, nullptr, e);
            lomljenje1(e->levi);
        }
        else{
            e->desni = new Vozlisce(aK, nullptr, nullptr, e);
            lomljenje1(e->desni);
        }
    }
}
```

Algoritem 5.7: Metoda *vstavi* preko parametra prejme celo število aK, ki ga nato vstavi v drevo.

5.2.5 Operacija odstranjevanja ključa iz drevesa

Za odstranitev poljubnega ključa iz lomljenega drevesa je definirana metoda *odstraniKoncno*, ki preko parametra prejme celo število aK. To število nato odstranimo iz drevesa. Postopek odstranjevanja vozlišča iz lomljenega drevesa je podrobneje opisan v razdelku 2.2.3.

```
void odstraniKoncno(int aK){
    if(jePrazno())
        return;

    Vozlisce* p = koren;
    Vozlisce* e;
    //iscemo kljuc aK v drevesu
    while{p != nullptr){
        e = p;
        /* ce kljuc najdemo, si shranimo njegovega starsa, odstranimo
        kljuc aK in, ce kljuc aK ni bil koren drevesa, izvedemo
        lomljenje nad njegovim starsem */
        if(p->kljuc == aK){
            e = p->stars;
            odstrani(aK);
            if(e != nullptr)
                lomljenje1(e);
            return;
        }
        //ce kljuca ne najdemo, se ustrezno premaknemo nivo nizje
        else if(aK < p->kljuc)
            p = p->levi;
        else
            p = p-> desni;
    }
    //ce kljuca aK ni v drevesu, izvedemo lomljenje na zadnjem
    //nenicelnem vozliscu, kjer priackujemo, da bi bil kljuc aK
    lomljenje1(e);
}
```

Algoritem 5.8: Metoda *odstraniKoncno* preko parametra prejme število aK, ki ga nato odstrani iz drevesa.

```
void odstrani(int aK){
    //poiscemo vozlisce s kljucem aK
    Vozlisce* v = najdi(aK);

    //v je list
    if(v->levi == nullptr && v-> desni == nullptr){
        //v je koren drevesa
        if(v == koren){
            delete v;
            koren = nullptr;
            return;
        }
        //v je levi otrok
        if(v->stars->levi == v)
            v->stars->levi = nullptr;
        //v je desni otrok
        else
            v->stars->desni = nullptr;
        delete v;
    }
    //v ima vsaj enega otroka
    else{
        //v ima samo levega otroka
        if(v->levi != nullptr && v->desni == nullptr){
            //v je koren drevesa
            if(v == koren){
                koren = v->levi;
                koren->stars = nullptr;
            }
            //v je levi otrok
            else if(v->stars->levi == v){
                v->stars->levi = v->levi;
                v->levi->stars = v->stars;
            }
            //v je desni otrok
            else{
                v->stars->desni = v->levi;
```

```
                v->levi->stars = v->stars;
            }
            delete v;
        }
        //v ima samo desnega otroka
        else if(v->levi == nullptr && v->desni != nullptr){
            // v je koren drevesa
            if(v == koren){
                koren = v->desni;
                koren->stars = nullptr;
            }
            //v je desni otrok
            else if(v->stars->desni == v){
                v->stars->desni = v->desni;
                v->desni->stars = v->stars;
            }
            //v je levi otrok
            else{
                v->stars->levi = v->desni;
                v->desni->stars = v->stars;
            }
            delete v;
        }
        //v ima oba otroka
        else{
            //poiscemo najvecji kljuc v levem poddrevesu od v
            Vozlisce* w = predhodnik(v);
            int k = w->kljuc;
            odstrani(w->kljuc);
            v->kljuc = k;
        }
    }
}
```

Algoritem 5.9: Metoda *odstrani* preko parametra prejme število aK, ki ga nato odstrani iz drevesa.

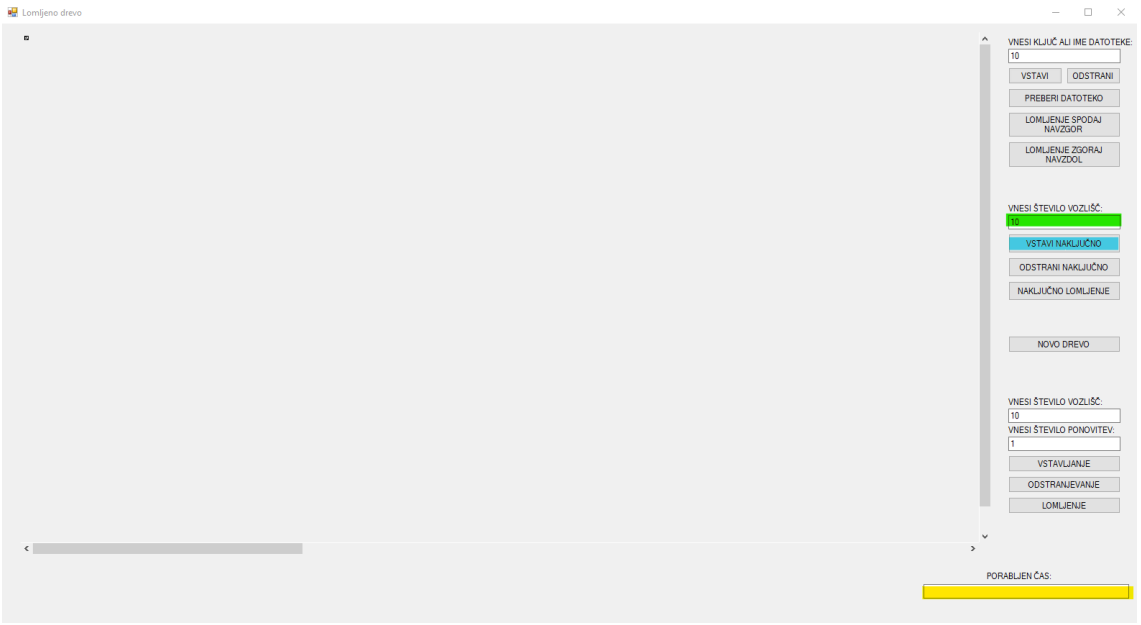
Opomba 5.7 *Metoda odstrani vozlišče odstrani iz drevesa enako kot v dvojiškem iskalnem drevesu.*

5.3 Metode z naključno izbiro vozlišč

V programu so definirane tudi metode za vstavljanje (glej tudi razdelek 5.1) in odstranjevanje naključno izbranih vozlišč v oz. iz drevesa ter za izvedbo lomljenja od spodaj navzgor nad naključno izbranimi vozlišči.

5.3.1 Vstavljanje

V vnosno polje, ki je na sliki 5.3 označeno z zeleno barvo, zapišemo število vozlišč, ki jih želimo vstaviti v drevo in pritisnemo gumb VSTAVI NAKLJUČNO. Recimo, da smo v polje vnesli vrednost n. Če je drevo na začetku prazno, potem v naključnem vrstnem redu v drevo vstavimo ključe od 1 do n. Če drevo na začetku ni prazno (recimo, da ima največji ključ enak k), potem v drevo v naključnem vrstnem redu vstavimo ključe od $k+1$ do $k+n$. Ko se metoda zaključi, se v polju PORABLJEN ČAS, na sliki 5.3 označeno z rumeno barvo, izpiše čas trajanja metode. Čas se meri v mikrosekundah.

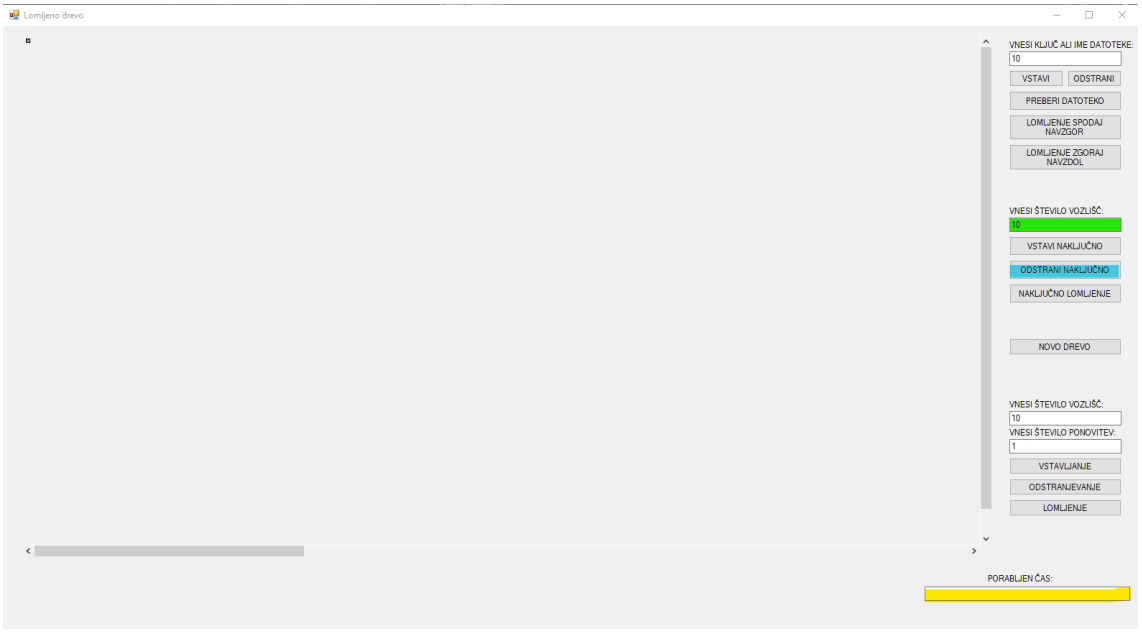


Slika 5.3: V vnosno polje zapišemo število vozlišč in pritisnemo gumb VSTAVI NAKLJUČNO.

5.3.2 Odstranjevanje

V vnosno polje, ki je na sliki 5.4 označeno z zeleno barvo, zapišemo število vozlišč, ki jih želimo odstraniti iz drevesa in pritisnemo gumb ODSTRANI NAKLJUČNO. Recimo,

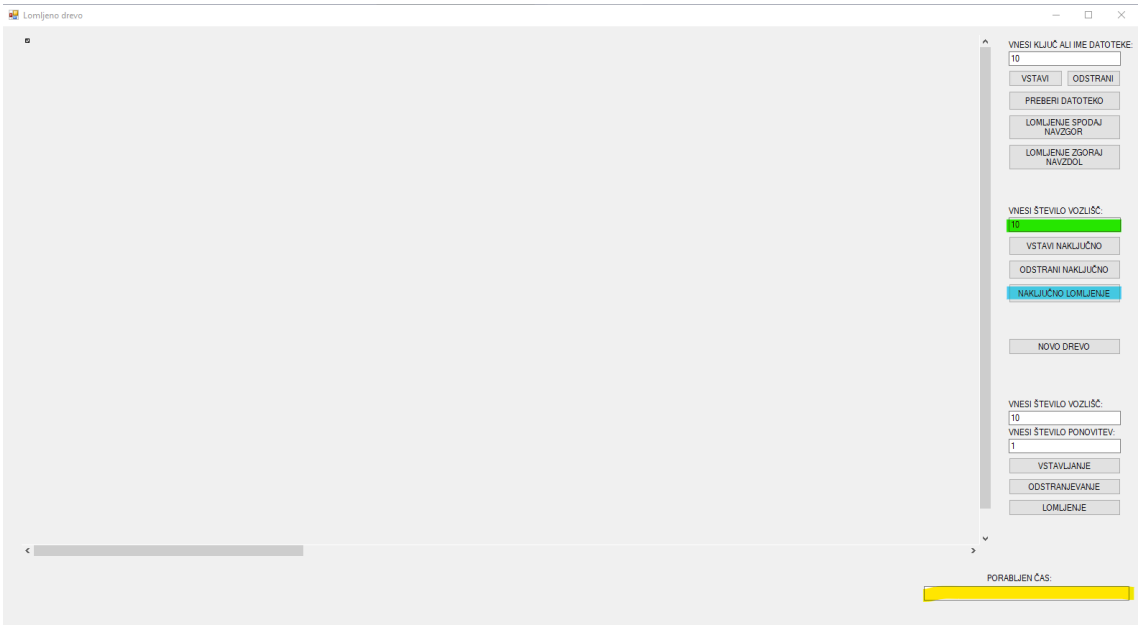
da smo vnesli vrednost n. Najprej poiščemo največji ključ v drevesu (recimo, da je to k) in nato v naključnem vrstnem redu iz drevesa odstranimo n vozlišč med 1 in k. Če nekega ključa ni v drevesu, potem samo izvedemo lomljenje od spodaj navzgor nad zadnjim neničelnim vozliščem, kjer pričakujem, da leži iskani ključ. Po končani operaciji se v polju PORABLJEN ČAS, ki je na sliki 5.4 označeno z rumeno barvo, izpiše čas trajanja metode. Čas se meri v mikrosekundah.



Slika 5.4: V vnosno polje zapišemo število vozlišč in pritisnemo gumb ODSTRANI NAKLJUČNO.

5.3.3 Lomljenje

V vnosno polje, ki je na sliki 5.5 označeno z zeleno barvo, zapišemo vrednost, nad koliko vozlišči želimo izvesti lomljenje od spodaj navzgor. Recimo, da smo vnesli vrednost n. Najprej poiščemo največji ključ v drevesu (recimo, da je to k) in nato n-krat izvedemo lomljenje od spodaj navzgor nad naključno izbranimi vozlišči med 1 in k. Če nekega ključa ni v drevesu, potem izvedemo lomljenje nad zadnjim neničelnim vozliščem, kjer pričakujemo, da leži želeni ključ. Po končani operaciji se v polju PORABLJEN ČAS, ki je na sliki 5.5 označeno z rumeno barvo, izpiše čas trajanja metode. Čas se meri v mikrosekundah.



Slika 5.5: V vnosno polje zapišemo število vozlišč in pritisnemo gumb NAKLJUČNO LOMLJENJE.

5.4 Analiza delovanja operacij

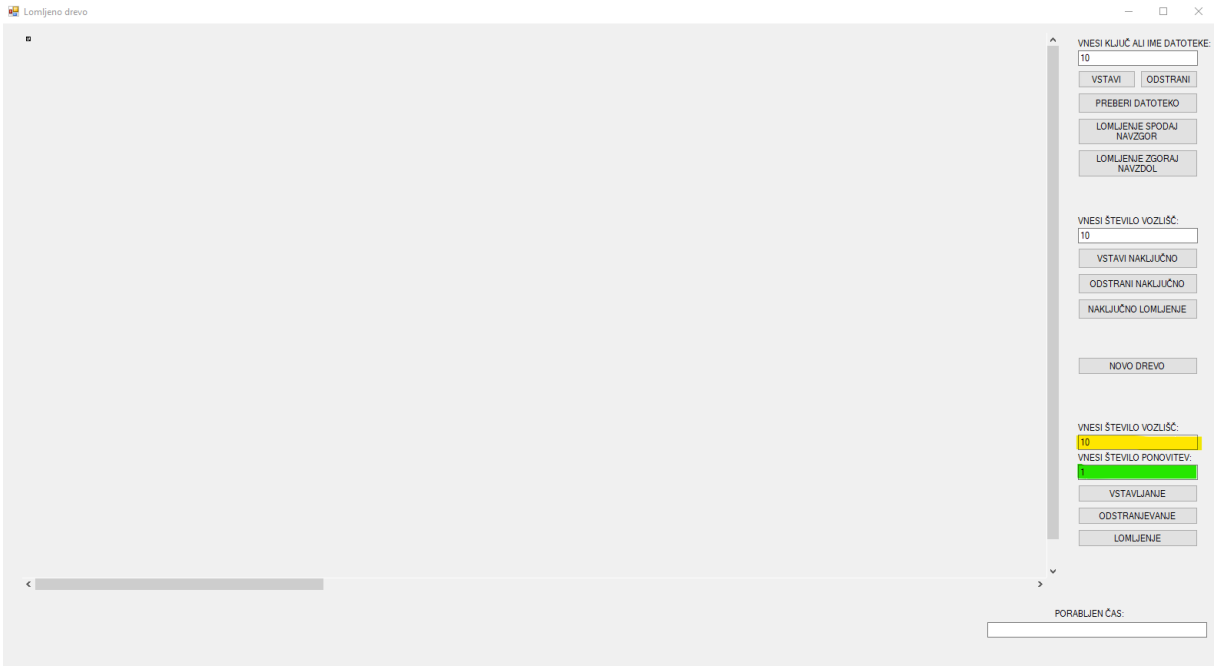
Za analizo implementiranih operacij na lomljenih drevesih, v programu definiramo operacije vstavljanja in odstranjevanja naključno izbranega ključa v oz. iz drevesa ter operacijo lomljenja na naključno izbranem vozlišču, kjer lahko te operacije izvedemo poljubno mnogokrat. Tako lahko lažje analiziramo hitrost delovanja omenjenih operacij.

V vnosno polje, ki je na sliki 5.6 označeno z rumeno barvo, zapišemo število vozlišč, ki jih želimo vstaviti v drevo, odstraniti iz drevesa ali izvesti lomljenje nad toliko vozlišči. Recimo, da smo v to polje zapisali vrednost n. V vnosno polje, ki je na sliki 5.6 označeno z zeleno, zapišemo kolikokrat želimo izvesti operacijo. Recimo, da smo v to polje zapisali vrednost i.

1. Če pritisnemo gumb VSTAVLJANJE, potem v drevo v naključnem vrstnem redu vstavimo vozlišča od 1 do n. Postopek ponovimo i-krat (v vsakem koraku ustvarimo novo drevo). Ko se operacija zaključi, se izriše trenutno drevo, če je vozlišč manj ali enako 200, in se v polju PORABLJEN ČAS izpiše povprečni čas naključnega vstavljanja n vozlišč v drevo. Čas se meri v mikrosekundah.

2. Če pritisnemo gumb ODSTRANJEVANJE, potem najprej v drevo v naključnem vrstnem redu vstavimo ključe od 1 do n, potem pa v naključnem vrstnem redu ta vozlišča odstranimo iz drevesa. Postopek ponovimo i-krat. Ko se operacija zaključi, se v polju PORABLJEN ČAS izpiše povprečni čas naključnega odstranjevanja n vozlišč iz drevesa. Čas se meri v mikrosekundah.

3. Če pritisnemo gumb LOMLJENJE, potem najprej v naključnem vrstnem redu v drevo vstavimo ključe od 1 do n in nato nad n naključno izbranimi vozlišči izvedemo operacijo lomljenje od spodaj navzgor (naključno izbiramo ključe med 1 in n, kjer se lahko posamezen ključ večkrat ponovi). Postopek ponovimo i-krat. Po končani operaciji se izriše trenutno drevo, če je vozlišč manj ali enako 200. V polje PORABLJEN ČAS se izpiše povprečni čas trajanja operacije lomljenje nad n naključno izbranih vozliščih. Čas se meri v mikrosekundah.



Slika 5.6: Začetno okno ob zagonu programa.

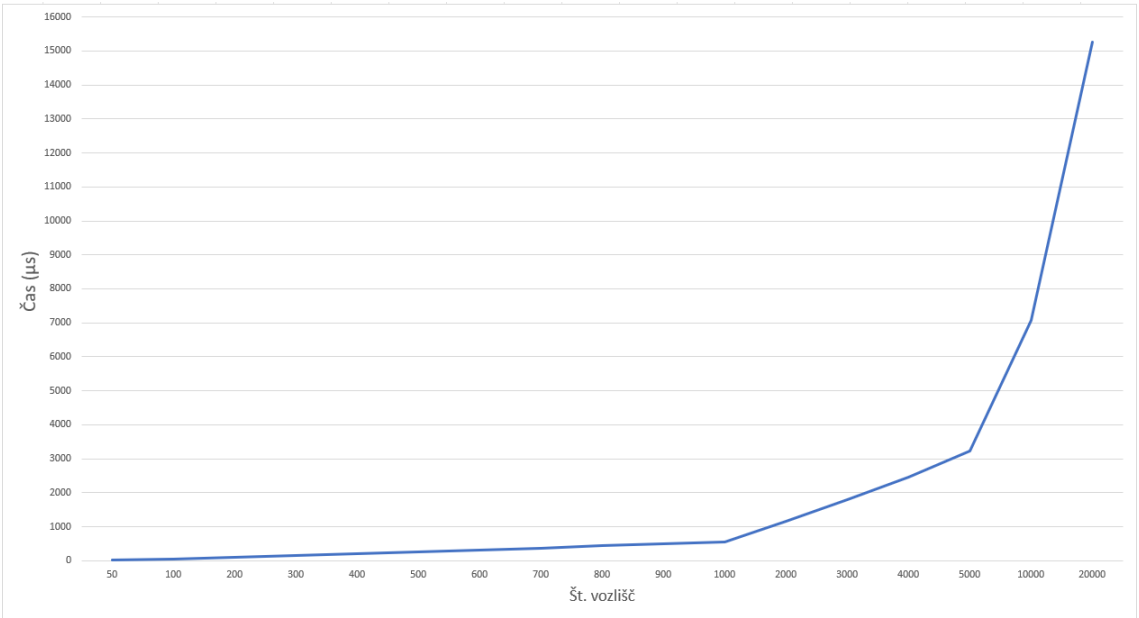
Recimo, da 1000-krat ponovimo vstavljanje n vozlišč v naključnem vrstnem redu v drevo. Potem program za različno število vozlišč izračuna povprečni čas trajanja operacije. V tabeli 5.1 so zapisani povprečni časi trajanja operacije vstavljanje za različne vrednosti števila vozlišč.

št. vozlišč (n)	čas (μs)
50	28,101
100	48,021
200	97,537
300	146,25
400	206,834
500	255,835
600	310,199

700	366,954
800	433,012
900	487,006
1000	547,214
2000	1163,445
3000	1807,541
4000	2466,229
5000	3221,045
10000	7063,307
20000	15266,091

Tabela 5.1: V tabeli so zapisani povprečni časi operacije vstavljanje glede na število vozlišč.

Če vrednosti iz tabele 5.1 predstavimo grafično, potem na sliki 5.7 vidimo, da se z večanjem števila vozlišč povečuje tudi čas trajanja operacije.



Slika 5.7: Grafični prikaz časa trajanja operacije vstavljanje glede na število vozlišč.

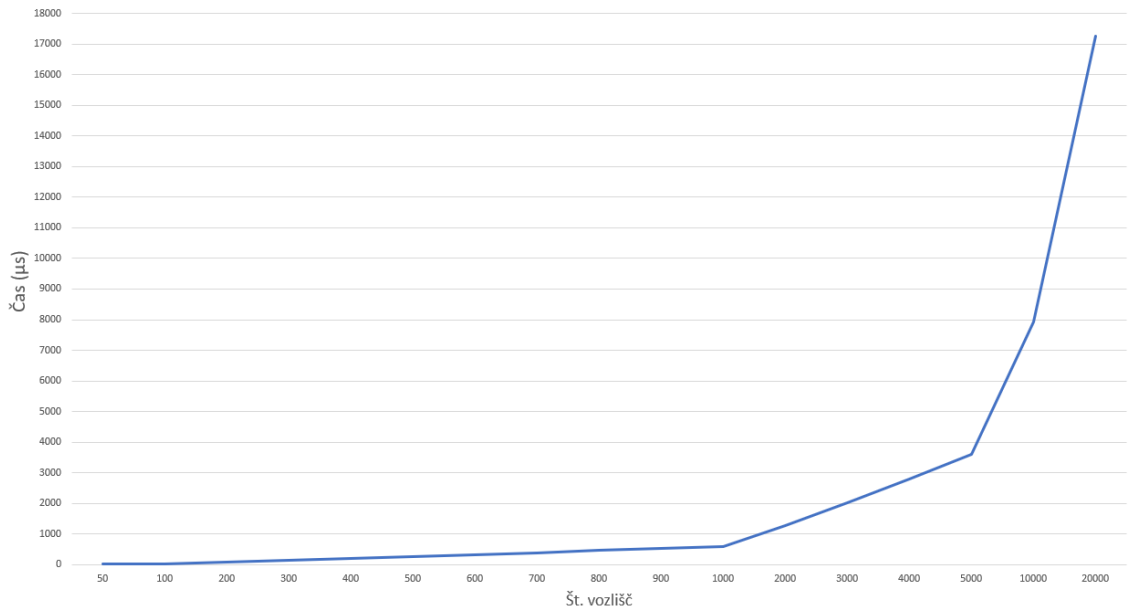
Če 1000-krat ponovimo naključno odstranjevanje n vozlišč iz drevesa, potem za različne vrednosti n dobimo povprečne čase zapisane v tabeli 5.2. Program je implementiran tako, da najprej v naključnem vrstnem redu vstavi n vozlišč v drevo in nato v naključnem vrstnem redu ta vozlišča odstrani iz drevesa. Ta postopek se ponovi 1000-krat. Na koncu se izpiše

povprečni čas trajanja operacije odstranjevanja (povprečni čas se meri izključno za del kode, kjer se izvaja odstranjevanje).

št. vozlišč (n)	čas (μs)
50	18,765
100	35,448
200	87,982
300	150,643
400	209,014
500	265,881
600	328,066
700	395,782
800	455,810
900	529,508
1000	582,118
2000	1271,73
3000	2035,112
4000	2798,544
5000	3613,408
10000	7921,779
20000	17252,110

Tabela 5.2: V tabeli so zapisani povprečni časi operacije odstranjevanja glede na število vozlišč.

Ponovno lahko vidimo na sliki 5.8, da se z večanjem števila vozlišč, povečuje tudi čas potreben za odstranitev vozlišč iz drevesa.



Slika 5.8: Grafični prikaz časa trajanja operacije odstranjevanja glede na število vozlišč.

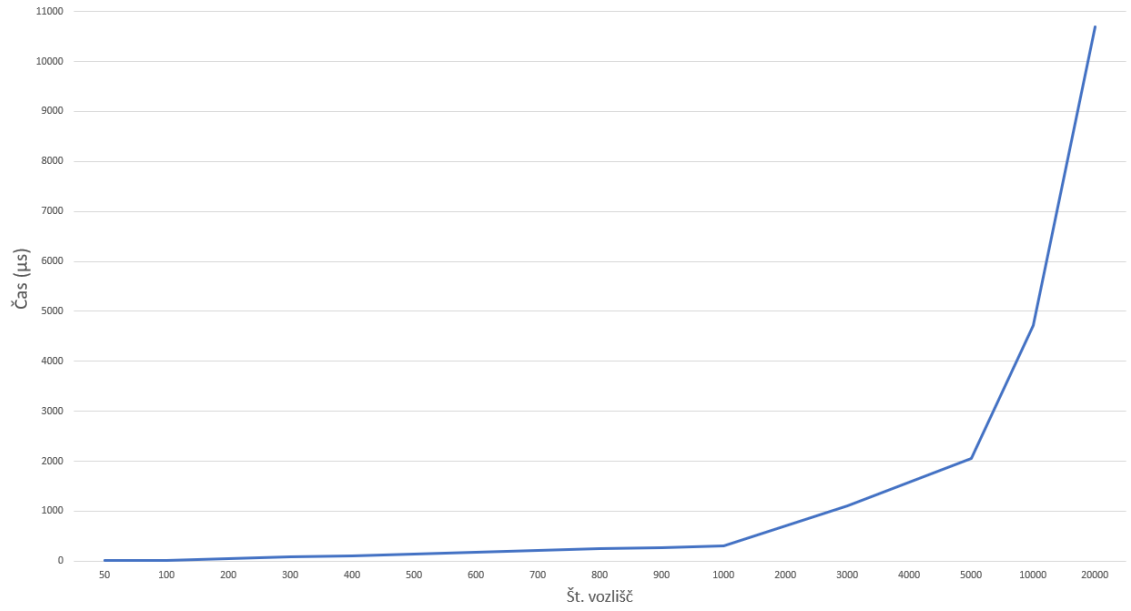
Če 1000-krat ponovimo operacijo lomljenje na naključno izbranih n vozliščih, potem za različne vrednosti n dobimo povprečne čase zapisane v tabeli 5.3. Program je implementiran tako, da najprej v naključnem vrstnem redu vstavi n vozlišč v drevo in nato izvede lomljenje nad naključno izbranem vozlišču. V vsaki ponovitvi se lomljenje izvede nad n vozlišči (vozlišča se lahko v tem primeru ponavljajo). Postopek ponovimo 1000-krat. Na koncu se izpiše povprečni čas trajanja operacije lomljenja (povprečni čas se meri izključno za del kode, kjer se izvaja lomljenje).

št. vozlišč (n)	čas (μs)
50	10,755
100	22,382
200	48,015
300	78,194
400	106,441
500	138,924
600	173,748
700	206,215
800	244,604
900	275,211
1000	310,478
2000	703,272

3000	1116,42
4000	1579,351
5000	2057,217
10000	4712,444
20000	10693,233

Tabela 5.3: V tabeli so zapisani povprečni časi operacije lomljenja glede na število vozlišč.

Če te podatke prikažemo grafično, kot prikazuje slika 5.9, vidimo, da se z večanjem števila vozlišč povečuje čas trajanja operacije lomljenje. Če primerjamo operacijo lomljenje z operacijama vstavljanje in odstranjevanje, opazimo, da je čas trajanja krajši, saj pri vstavljanju in odstranjevanju tudi uporabimo lomljenje in se s tem čas trajanja nekoliko poviša.



Slika 5.9: Grafični prikaz časa trajanja operacije lomljenja glede na število vozlišč.

Poglavje 6

Uravnotežena drevesa

Uravnoteženo dvojiško iskalno drevo poleg vzdrževanja vrstnega reda vozlišč, vzdržuje tudi njegovo višino, ki je $O(\log n)$ tudi po vstavljanju in odstranjevanju vozlišča iz drevesa. Da to doseže, se mora drevo ponovno uravnotežiti, ko vstavimo ali odstranimo vozlišče.

Obstaja več vrst uravnoteženih dreves, ki se med sabo razlikujejo v načinu vzdrževanja ravnotežja v drevesu. V nadaljevanju so na kratko predstavljena AVL drevesa, rdeče črna drevesa in B-drevesa ter primerjava teh dreves z lomljenimi drevesi.

6.1 AVL drevesa

Definicija 6.1 ([5]) *AVL drevo je dvojiško iskalno drevo, v katerem za vsako vozlišče x velja:*

1. *levo in desno poddrevo od x sta AVL drevesi,*
2. *globini levega in desnega poddrevesa od x se razlikujeta za največ 1.*

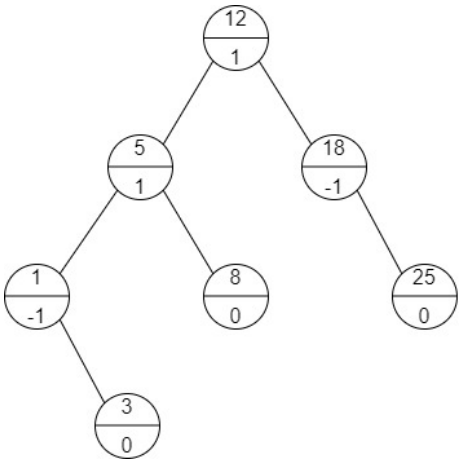
Za predstavitev AVL dreves uporabimo dinamično predstavitev opisano v razdelku 1.2.2, kjer vsako vozlišče vsebuje še podatek o ravnotežnem faktorju.

Definicija 6.2 ([5]) *Ravnotežni faktor $RF(x)$ vozlišča x je razlika med globino levega in desnega poddrevesa od x,*

$$RF(x) = globina(levi\ otrok_x) - globina(desni\ otrok_x).$$

V AVL drevesu za vsako vozlišče x velja, da je $RF(x) \in \{-1, 0, 1\}$.

Na sliki 6.1 je zgled drevesa, kjer je znotraj vsakega vozlišča pod ključem zapisan tudi ravnotežni faktor. Ker ima vsako vozlišče x ravnotežni faktor $RF(x) \in \{-1, 0, 1\}$, je to AVL drevo.



Slika 6.1: Primer dvojiškega iskalnega drevesa, ki je AVL drevo.

Vstavljanje in odstranjevanje vozlišča v AVL drevesu poteka enako kot pri dvojiških iskalnih drevesih (glej 1.3.1) le, da moramo na koncu operacije preveriti, ali ima vsako vozlišče x ravnotežni faktor $RF(x) \in \{-1, 0, 1\}$. Če ugotovimo, da katero izmed vozlišč nima ustreznega ravnotežnega faktorja, potem z uporabo rotacij (glej 2.1) drevo ponovno uravnotežimo. Časovna zahtevnost vstavljanja, odstranjevanja in iskanja vozlišča v AVL drevesih je enaka $O(\log(n))$.

Več o rotacijah, operacijah in zahtevnosti AVL dreves najdemo v [3, 5, 8].

6.1.1 Primerjava lomljenih drevesa z AVL drevesi

Tako lomljena kot AVL drevesa so vrsta uravnoteženih dvojiških iskalnih dreves z dobro časovno zahtevnostjo, vendar se razlikujejo po tem, kako to zahtevnost dosežejo. AVL drevesa imajo časovno zahtevnost operacij vstavljanja, odstranjevanja in iskanja vozlišča enako $O(\log(n))$, medtem ko lahko lomljena drevesa dosežejo enako zahtevnost, vendar to zahtevnost dosežejo le v amortiziranem smislu. Poljubno dolgo zaporedje operacij vstavljanja, odstranjevanja in iskanja vozlišča v lomljenih drevesih ima lahko amortizirano časovno zahtevnost največ $O(n \log(n))$.

V AVL drevesih je oblika drevesa ves čas omejena tako, da je uravnotežena, kar pomeni, da višina drevesa nikoli ne preseže $O(\log n)$. Ta oblika se ohrani tudi po operaciji vstavljanja

ali odstranjevanja vozlišča, medtem ko pri iskanju vozlišča, drevo ne spremeni oblike. Po drugi strani pa lomljena drevesa ohranjajo učinkovitost tako, da preoblikujejo drevo po vsaki operaciji ne glede na to ali gre za vstavljanje, odstranjevanje ali iskanje vozlišča. Tako so vozlišča, nad katerim pogosto izvajamo operacije, vedno blizu vrha in imajo boljšo časovno zahtevnost operacij. Oblika lomljenih dreves tako ni omejena in se razlikuje glede na izvedene operacije. Glede na prostorsko zahtevnost so lomljena drevesa učinkovitejša, saj jim v vozlišču ni potrebno shranjevati informacije o ravnotežnem faktorju.

6.2 Rdeče-črna drevesa

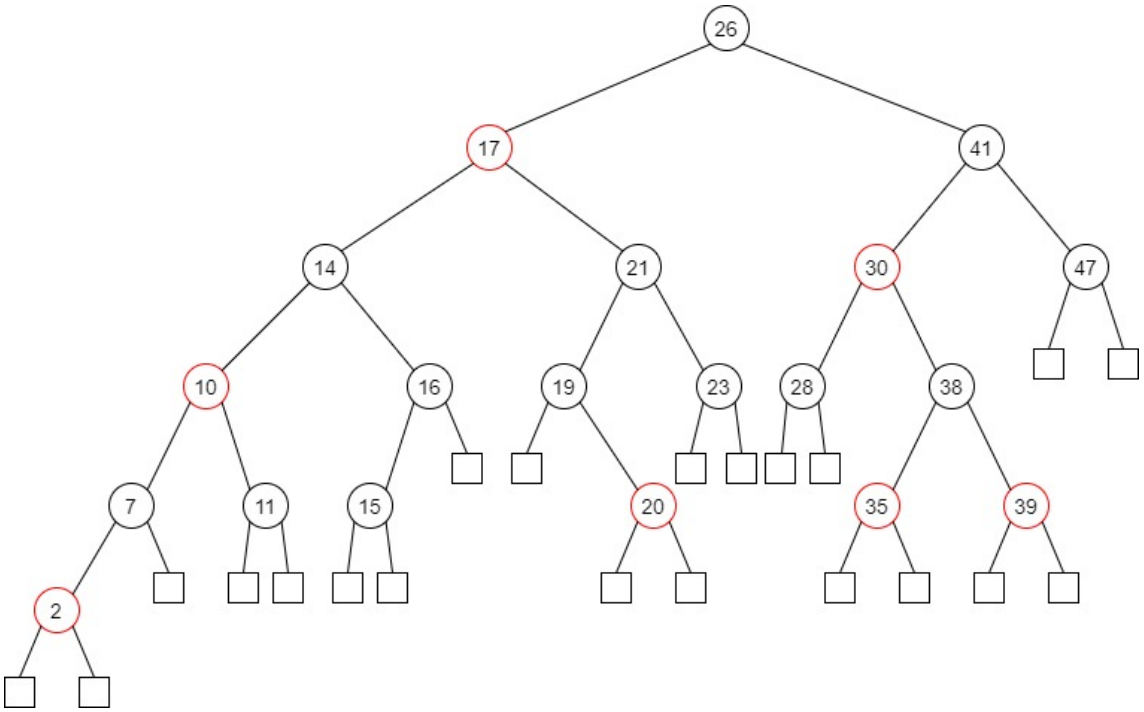
Rdeče-črna drevesa so razširjena drevesa, kar pomeni, da vsako vozlišče brez otrok (z enim otrokom) ima dva (enega) zunanja prazna otroka.

Definicija 6.3 ([5]) *Rdeče-črno drevo je dvojiško iskalno drevo, ki zadošča naslednjim lastnostim:*

1. *vsako vozlišče je bodisi rdeče, bodisi črne barve,*
2. *koren drevesa je črne barve,*
3. *vsako zunanje prazno vozlišče je črno,*
4. *če je vozlišče rdeče, sta oba otroka črna, in*
5. *najkrajša pot od nekega vozlišča do poljubnega lista ima vedno enako število črnih vozlišč.*

Za predstavitev rdeče-črnih dreves uporabimo dinamično predstavitev dreves predstavljeno v razdelku 1.2.2, kjer vsako vozlišče vsebuje še podatek o barvi.

Primer rdeče-črnega drevesa je prikazan na sliki 6.2.



Slika 6.2: Primer rdeče-črnega drevesa z zunanjimi praznimi vozlišči.

Vstavljanje in odstranjevanje vozlišča v rdeče-črno drevo poteka enako kot pri dvojiških iskalnih drevesih (glej 1.3.1) le, da moramo na koncu operacije morebiti popraviti barve vozlišč. Če ugotovimo, da barve niso ustrezne, problem rešimo z uporabo rotacij (glej 2.1) in menjavo barv vozlišč. Časovna zahtevnost vstavljanja, odstranjevanja in iskanja vozlišča v rdeče-črnih drevesih je enaka $O(\log(n))$.

Več o rdeče-črnih drevesih je predstavljeno v [1, 3].

6.2.1 Primerjava lomljenih drevesa z rdeče-črnimi drevesi

Tako lomljena kot rdeče-črna drevesa so vrsta uravnoteženih dvojiških iskalnih dreves. Časovno gledano so rdeče-črna drevesa hitra za operacije vstavljanja, odstranjevanja in iskanja vozlišča, saj imajo zahtevnost enako $O(\log n)$, katero lahko dosežejo tudi lomljena drevesa v t.i. amortiziranem smislu, saj smo dokazali, da ima lahko poljubno dolgo zaporedje operacij vstavljanja, odstranjevanja in iskanja vozlišča amortizirano časovno zahtevnost največ $O(n \log n)$.

6.3 B-drevesa

B-drevesa so oblika posplošitve dvojiškega iskalnega drevesa, kjer lahko ima vsako vozlišče več kot en ključ in posledično več kot dva otroka.

Definicija 6.4 ([1]) *B-drevo je iskalno drevo, za katerega veljajo naslednje lastnosti:*

1. *vsako vozlišče x vsebuje naslednje komponente:*
   - *$x.n$ predstavlja število ključev vozlišča x,*
   - *$x.n$ ključev vozlišča x, $x.kljuc_1, x.kljuc_2, \ldots, x.kljuc_{x.n}$, je shranjenih v urejenem vrstnem redu tako, da je*

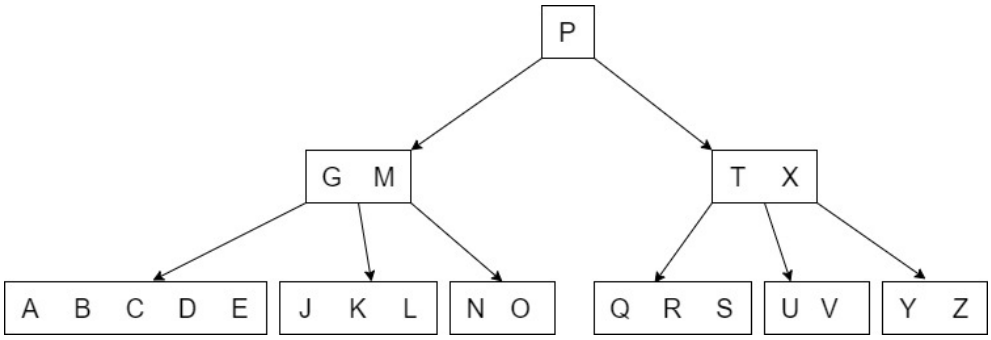
     $$x.kljuc_1 \leq x.kljuc_2 \leq \ldots \leq x.kljuc_{x.n},$$

   - *$x.list$, ki vsebuje vrednost $TRUE$, če je vozlišče x list drevesa in $FALSE$ sicer.*
2. *če je x notranje vozlišče, potem vsebuje $x.n + 1$ kazalcev, $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$, na svoje otroke,*
3. *ključi $x.kljuc_i$ ločujejo intervale ključev v vsakem poddrevesu: če je k_i poljuben ključ v poddrevesu s korenom $x.c_i$, potem je*

   $$k_1 \leq x.kljuc_1 \leq k_2 \leq x.kljuc_2 \leq \ldots \leq x.kljuc_{x.n} \leq k_{x.n+1},$$

4. *vsi listi drevesa so na enakem nivoju h,*
5. *ključi imajo spodnjo in zgornjo mejo glede na to koliko ključev lahko vsebujejo. Te meje določimo glede na fiksno naravno število $t \geq 2$, ki ga imenujemo* najmanjša stopnja B-drevesa*:*
   - *vsako vozlišče razen korena mora imeti vsaj $t - 1$ ključev. Tako ima vsako notranje vozlišče razen korena vsaj t otrok. Če drevo ni prazno, potem mora koren imeti vsaj en ključ,*
   - *vsako vozlišče lahko vsebuje največ $2t - 1$ ključev. Potemtakem lahko ima notranje vozlišče največ $2t$ otrok. Pravimo, da je vozlišče* polno*, če vsebuje $2t - 1$ ključev.*

Primer B-drevesa je prikazan na sliki 6.3. Drevo je stopnje 3.



Slika 6.3: Primer B-drevesa stopnje 3.

Iskanje vozlišča v B-drevesu poteka podobno kot pri dvojiških iskalnih drevesih. Razlika je v tem, da se med iskanjem v dvojiškem iskalnem drevesu premaknemo bodisi v levo, bodisi v desno poddrevo. Pri B-drevesih tako imamo pri vsakem vozlišču x, namesto dveh možnih premikov navzdol, $x.n+1$ možnih premikov navzdol po drevesu. Časovna zahtevnost iskanja v B-drevesih znaša $O(\log n)$, kjer je n število vozlišč drevesa.

Vstavljanje novega ključa v B-drevesa je kompleksnejše od vstavljanja ključa v dvojiško iskalno drevo. Novi ključ vedno vstavimo v že obstoječi list drevesa. Če je list poln (ima $2t-1$ ključev), potem uporabimo operacijo *medianski ključ*, ki polno vozlišče razbije na dve novi vozlišči (vsako z $t-1$ ključi), medianski ključ pa se premakne v starševsko vozlišče. Če je drevo sestavljeno samo iz korenskega vozlišča, potem to vozlišče razbijemo na dve novi vozlišči (vsako z $t-1$ ključi) in medianski ključ postane novi koren drevesa. Časovna zahtevnost operacije vstavljanja v B-drevesih je $O(\log n)$, kjer je n število vozlišč drevesa.

Pri odstranjevanju ključa iz B-drevesa lahko ključ odstranimo iz kateregakoli vozlišča. Če odstranimo ključ iz notranjega vozlišča, potem moramo prerazporediti otroke vozlišča iz katerega smo odstranili ključ. Pri odstranjevanju moramo paziti, da ima vsako vozlišče razen korena vsaj $t-1$ ključev. Okvirna rešitev je, da ključ iz starševskega vozlišča potisnemo navzdol v njegovega otroka. Časovna zahtevnost odstranjevanja ključa iz B-drevesa je $O(\log n)$, kjer je n število vozlišč drevesa.

Več o B-drevesih je predstavljeno v [1, 5].

6.3.1 Primerjava lomljenih dreves z B-drevesi

Če primerjamo lomljena drevesa z B-drevesi ugotovimo, da se med sabo razlikujeta že po sami strukturi. Vozlišča v lomljenem drevesu lahko imajo največ dva otroka, medtem ko imajo B-drevesa teh lahko več. Če je B-drevo stopnje m, potem lahko ima vsako vozlišče največ $2m$ otrok. Vozlišča B-drevesa lahko imajo tudi več kot en ključ, medtem ko lomljena drevesa lahko imajo natanko enega. Listi lomljenega drevesa so lahko na različnih nivojih,

medtem ko pri B-drevesih morajo biti na enakem nivoju. Če vstavljamo ključ v lomljeno drevo, potem je operacija vstavljanja enostavnejša od operacije vstavljanja ključa v B-drevo, saj moramo še preveriti in popraviti drevo tako, da zadošča vsem pogojem iz definicije B-drevesa.

Literatura

[1] T. H. Cormen, C.E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms, Third Edition*, MIT Press, 2009.

[2] J. Erickson, *Algorithms*, University of Illinois at Urbana-Champaign, 1999.

[3] J. Kozak, *Podatkovne strukture in algoritmi*, Društvo matematikov, fizikov in astronomov, Ljubljana, 1986.

[4] D. C. Kozen, *The design and analysis of algorithms*, Springer Science & Business Media, 1992.

[5] R. L. Kruse, A. J. Ryba, *Data Structures and Program Design in C++*, Prentice Hall, 2000.

[6] A. Ozdemir, *Top-Down/Bottom-Up Splay Duality and Generalized Top-Down Splay*, Harvey Mudd College, 2017.

[7] H. A. Sayed, *The Complexity of Splay Trees and Skip Lists*, Diss. University of the Western Cape, 2008.

[8] C. A. Shaffer, *Data structures and algorithm analysis. Edition 3.2 (C++ Version)*, 2011.

[9] D. D. Sleator, R. E. Tarjan, *Self-adjusting binary search trees, Journal of the ACM*, 32(3):652-686, 1985.

[10] Design and Analysis of Algorithms, *Amortized analysis*, Lecture 11. Massachusetts Institute of Technology: MIT OpenCouseWare. Dostopano na naslovu: `https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2012/resources/mit6_046js12_lec11/` (datum dostopa: 25. 11. 2021)

[11] Chapter 18: Amortized analysis. Dostopano na naslovu: `http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap18.htm` (datum dostopa: 25. 11. 2021)

[12] Lecture 18: Amortyzed analysis. Dostopano na naslovu: http://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec20-amortized/amortized.htm (datum dostopa: 25. 11. 2021)