

UNIVERZA NA PRIMORSKEM  
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN  
INFORMACIJSKE TEHNOLOGIJE

Magistrsko delo  
**Kompaktna priponska drevesa**  
(Compact Suffix Tree)

Ime in priimek: *Jani Suban*

Študijski program: *Računalništvo in informatika, 2. stopnja*

Mentor: *prof. dr. Andrej Brodnik*

Somentor: *izr. prof. dr. Rok Požar*

Koper, 2025

## Ključna dokumentacijska informacija

Ime in PRIIMEK: Jani SUBAN

Naslov magistrskega dela: Kompaktna priponska drevesa

Kraj: Koper

Leto: 2025

Število listov: 87

Število slik: 22

Število tabel: 4

Število referenc: 31

Mentor: prof. dr. Andrej Brodnik

Somentor: izr. prof. dr. Rok Požar

UDK:

Ključne besede:

Priponska drevesa, Priponska polja, Kompaktna priponska drevesa, Kompaktne podatkovne strukture

Izvleček:

Magistrska naloga obravnava problem iskanja vzorcev v daljših besedah z uporabo podatkovnih struktur, kot so priponsko drevo, priponsko polje in kompaktno priponsko drevo. Namen naloge je preučiti vpliv kompaktne predstavitve na izgradnjo in uporabo priponskih dreves, zato je bila storjena empirična primerjava teh struktur. Primerjava je bila storjena za zaporedje genov ter za besedilo v naravnem jeziku. Pri tem se vidijo praktične prednosti kompaktnosti, saj se zmanjša učinkovitost priponskega drevesa, ko preseže velikost delovnega pomnilnika in zato potrebuje Swap razdelek na disku. Te težave nimata priponsko polje in kompaktno priponsko drevo za testirane velikosti besed.

## Key document information

Name and SURNAME: Jani SUBAN

Title of the thesis: Compact Suffix Tree

Place: Koper

Year: 2025

Number of pages: 87

Number of figures: 22

Number of tables: 4

Number of references: 31

Mentor: prof. dr. Andrej Brodnik

Co-Mentor: izr. prof. dr. Rok Požar

UDC:

Keywords:

Suffix tree, Suffix array, Compressed suffix tree, compact data structures

Abstract:

The master's thesis addresses the problem of pattern matching in longer texts using data structures such as the suffix tree, suffix array, and compressed suffix tree. The aim of the thesis is to examine the impact of compact representation on the construction and use of suffix trees, for which an empirical comparison of these structures was performed. The comparison was carried out on a gene sequence and on a natural language text. The practical advantages of compactness are evident, as the efficiency of the suffix tree decreases when it exceeds the size of the main memory and thus requires the use of a swap partition on the disk. The suffix array and the compressed suffix tree did not face this issue with the tested text sizes.

# Kazalo vsebine

<b>1</b>	<b>UVOD</b>	<b>1</b>
1.1	STRUKTURA . . . . .	1
<b>2</b>	<b>OZNAKE, DEFINICIJE IN OSNOVNE OPERACIJE</b>	<b>3</b>
2.1	OZNAKE IN OSNOVNE DEFINICIJE . . . . .	3
2.2	ENIŠKI ZAPIS . . . . .	4
2.3	MODEL RAČUNANJA . . . . .	4
2.4	BITNO POLJE . . . . .	5
2.5	KOMPAKтна PREDSTAVITEV DREVES . . . . .	11
2.5.1	Zaporedje eniških zapisov stopenj vozlišč po plasteh . . . . .	12
2.5.2	Zaporedje eniških zapisov stopenj vozlišč v globino . . . . .	14
2.5.3	Uravnoteženi oklepaji . . . . .	16
<b>3</b>	<b>PRIPONSKA DREVESA</b>	<b>19</b>
3.1	POIZVEDBE . . . . .	20
3.2	GRADNJA . . . . .	23
3.2.1	Naivna metoda . . . . .	23
3.2.2	Izboljšana naivna metoda . . . . .	26
3.2.3	Ukkonenov algoritem . . . . .	28
<b>4</b>	<b>PRIPONSKO POLJE</b>	<b>33</b>
4.1	POIZVEDBE . . . . .	34
4.2	GRADNJA . . . . .	38
4.3	SIMULACIJA OPERACIJ PRIPONKEGA DREVESA . . . . .	40
<b>5</b>	<b>KOMPAKтна PREDSTAVITEV PRIPONKEGA DREVESA</b>	<b>46</b>
5.1	POIZVEDBE . . . . .	54
5.2	GRADNJA . . . . .	56
<b>6</b>	<b>EMPIRIČNA EVALVACIJA</b>	<b>60</b>
6.1	EKSPERIMENTALNO OKOLJE . . . . .	60
6.1.1	Testni primeri . . . . .	60
6.1.2	Metoda testiranja . . . . .	62

6.1.3	Stroj in operacijski sistem . . . . .	62
6.2	REZULTATI . . . . .	64
6.2.1	Gradnja . . . . .	65
6.2.2	Poizvedbe . . . . .	67
6.3	RAZPRAVA . . . . .	70
<b>7</b>	<b>ZAKLJUČEK</b>	<b>73</b>
<b>8</b>	<b>LITERATURA IN VIRI</b>	<b>75</b>

# Kazalo preglednic

1	Implementacija operacij drevesa v LOUDS. . . . .	13
2	Implementacija operacij drevesa z DFUDS . . . . .	15
3	Implementacija operacij drevesa z BP . . . . .	17
4	Primerjava besedil, ki smo jih uporabili za primerjavo različnih indeksov besed . . . . .	61

# Kazalo slik in grafikonov

1	Primer dodatne podatkovne strukture za operacijo $\text{rang}_1$ v konstantnem času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine $w = 4$ . Pri tem se je izbralo, da je število $k = 4$ . . . . .	7
2	Primer dodatne podatkovne strukture za operacijo $\text{izbira}_1$ v $O(\log \log n)$ času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine $w = 4$ . Zaradi majhnosti primera velika vedra, ki vsebujejo vsaj 10 bitov namesto $s \log^2 b = 4 \log^2 36$ bitov, kar se zaokroži na 100 bitov. Velika vedra so označena v bitnem polju s sivo barvo. . . . .	8
3	Primer predstavitve drevesa z metodo LOUDS za priponsko drevo besede »KOKOŠ\$«. . . . .	13
4	Primer predstavitve drevesa z metodo DFUDS za priponsko drevo besede »KOKOŠ\$«. . . . .	15
5	Primer predstavitve drevesa z metodo BP za priponsko drevo besede »KOKOŠ\$«. . . . .	17
6	Primer priponskega drevesa nad besedo »KOKOŠ\$«. . . . .	20
7	Načini dodajanja novih znakov v priponsko drevo. Na sliki kvadrati predstavljajo liste drevesa, krogi pa predstavljajo notranja vozlišča drevesa. . . . .	24
8	Primer gradnje priponskega drevesa z uporabo Naivne metode za besedo »KOKOŠ\$«. . . . .	25
9	Primer gradnje priponskega drevesa z uporabo Izboljšane naivne metode za besedo »KOKOŠ\$«. . . . .	27
10	Primer gradnje priponskega drevesa z uporabo Ukkonenovega algoritma za besedo »KOKOŠ\$«. . . . .	29
11	Primer priponskega drevesa in priponskega polja z dodatnim $LCP$ poljem nad besedo »KOKOŠ\$«. . . . .	33
12	Prikaz razmerja med $P$ in priponama $T[SA[L] : ]$ in $T[SA[M] : ]$ . . . .	35
13	Primer $L\text{-}LCP$ in $R\text{-}LCP$ polji za priponsko polje nad besedo »KOKOŠ\$«. . . . .	37
14	Primer $LCP$ intervalnega drevesa nad priponskim poljem in $LCP$ poljem besede »KOKOŠ\$« ter njegova predstavitev s tabelo. Na sliki je dodano tudi priponsko drevo besede »KOKOŠ\$«. . . . .	42

15	Primer komponent kompaktne priponskega drevesa nad besedo »KOKOŠ\$«.	53
16	Posnetek zaslona upravljalnika opravil Htop med izgradnjo priponskega drevesa za besedilo dolžine 2048000 znakov. . . . .	65
17	Graf velikosti indeksov izgrajenih iz besed različnih velikosti. Vhodna beseda je DNKSEKVENCA. . . . .	66
18	Graf velikosti indeksov izgrajenih iz besed različnih velikosti. Vhodna beseda je roman NAKLANCU. . . . .	67
19	Graf prikazuje čas gradnje indeksov besede za različne dolžine vhodnih besed. Vhodna beseda je DNKSEKVENCA. . . . .	68
20	Graf prikazuje čas gradnje indeksov besede za različne dolžine vhodnih besed. Vhodna beseda je NAKLANCU. . . . .	69
21	Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih indeksih besede. Vhodna beseda je DNKSEKVENCA. . . . .	70
22	Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih indeksih besede. Vhodna beseda je NAKLANCU. . . . .	71



# Seznam kratic

angl.	Angleščina
slo.	Slovenščina
idr.	in drugi
DNK	Deoksiribonukleinska kislina
KMP	Knuth–Morris–Pratt
ST	angl. <i>Suffix Tree</i> (slo. Priponsko drevo)
rmM	angl. <i>range minimum-maximum</i>
rmq	angl. <i>range minimum query</i>
rMq	angl. <i>range maximum query</i>
LOUDS	angl. <i>Level Order Unary Degree Sequence</i> (slo. Zaporedje eniških zapisov stopenj vozlišč po plasteh)
DFUDS	angl. <i>Depth First Unary Degree Sequence</i> (slo. Zaporedje eniških zapisov stopenj vozlišč v globino)
BP	angl. <i>Balanced Parentheses</i> (slo. Uravnoreženi oklepaji)
SA	angl. <i>Suffix Array</i> (slo. Priponsko Polje)
LCP	angl. <i>Longest Common Prefix</i> (slo. Najdaljša skupna predpona)
CST	angl. <i>Compressed Suffix Tree</i> (slo. Kompaktno priponsko drevo)
CSA	angl. <i>Compressed Suffix Array</i> (slo. Kompaktno priponsko polje)
GCC	angl. <i>GNU C Compiler</i> (slo. GNU C Prevajalnik)
SDSL	angl. <i>Succinct Data Structure Library</i> (slo. Knjižnica kompaktnih podatkovnih struktur)
ASCII	angl. <i>American Standard Code for Information Interchange</i> (slo. Ameriški standardni nabor za izmenjavo informacij)

## Zahvala

Najprej bi se rad zahvalil mentorju prof. dr. Andreju Brodniku in somentorju izr. prof. dr. Roku Požarju za pomoč pri izdelavi magistrske naloge. Poleg tega bi se rad zahvalil tudi svoji družini in prijateljem. Posebno pa bi se rad zahvalil svojim staršem in bratu, ki so mi bili v pomoč ter mi stali ob strani v času študija.

# 1 UVOD

Pogosti problem pri obdelavi daljših besed je iskanje vzorcev v njih. V procesiranju naravnih jezikov vlogo vzorca pogosto prevzame iskana beseda naravnega jezika, vlogo vhodne besede pa prevzame besedilo, v katerem želimo iskati besedo. Medtem ko v bioinformatiki vlogo vzorca prevzame specifični gen ali druga zaporedja DNK, pri tem pa je vhodna beseda genom.

Ko v besedi  $T$  dolžine  $n$  iščemo zgolj en vzorec  $P$  dolžine  $m$ , se za iskanje lahko uporabi Knuth–Morris–Pratt (KMP) algoritem s časovno zahtevnostjo  $O(n + m)$ . V primeru, da se v besedi išče več vzorcev, je smiselno nad besedo zgraditi indeks, recimo priponsko drevo, ki je lahko zgrajeno v času  $O(n)$ , in nato iskati vzorce. Časovna zahtevnost iskanja vzorca v priponskem drevesu je sorazmerna dolžini vzorca,  $O(m)$ . Ko je vzorec prisoten v besedi, se vsaj ena pripona besede začne z iskanim vzorcem. Torej se vzorec zagotovo nahaja na začetku sprehoda iz korena proti listom priponskega drevesa, zgrajenega nad besedo. Peter Weiner je kot prvi predstavil priponska drevesa leta 1973 [5] in jih uporabil v algoritmu za iskanje vzorcev v besedi. Časovna zahtevnost podanega algoritma za iskanje vzorcev v besedi je  $O(m)$ . Velikost priponskega drevesa je sorazmerna z dolžino vhodne besede, torej pri velikih besedah preraste velikost delovnega pomnilnika, kar močno vpliva na hitrost iskanja. Rešitev te težave je kompaktna predstavitev priponskega drevesa [6].

Namen magistrske naloge je preučevanje vpliva kompaktne predstavitve na izgradnjo priponskih dreves in na operacije nad njimi. V nalogi bodo predstavljene časovne zahtevnosti osnovnih operacij in poizvedb ter izgradnje priponskih dreves, priponskih polj in kompaktne predstavitve priponskih dreves. Izdelana bo tudi empirična primerjava različnih implementacij priponskih dreves in drugih indeksov besed.

## 1.1 STRUKTURA

Magistrska naloga je razdeljena na pet delov. V drugem poglavju bodo predstavljene notacije, ki so uporabljene v magistrski nalogi. Predstavljene bodo tudi osnovne podatkovne strukture in operacije nad njimi, ki bodo uporabljene za implementacijo kompaktne predstavitve priponskega drevesa.

V tretjem poglavju naloge bo podrobno predstavljena podatkovna struktura priponsko drevo (angl. *Suffix Tree* oziroma ST). Poleg predstavitve strukture bo v poglavju

prikazana tudi metoda, ki sprotno (angl. *on-line*) gradi priponsko drevo v času  $O(n)$  ter implementacija poizvedb nad vhodno besedo s pomočjo priponskega drevesa. Zatem sledi v četrtem poglavju predstavitev alternativnega indeksa besedila, priponsko polje (angl. *Suffix Array* oziroma SA). Predstavljena bo tudi metoda izgradnje priponskega polja v  $O(n)$  času ter implementacija poizvedb s priponskim poljem.

V petem poglavju bo predstavljena podatkovna struktura kompaktno priponsko drevo (angl. *Compressed Suffix Tree* oziroma CST), ki omogoča iste operacije kot priponsko drevo, pri tem pa potrebuje manj prostora. V tem poglavju bo tudi predstavljen algoritem za izgradnjo kompaktnega priponskega drevesa, ki skozi celotno izgradnjo ohranja kompaktnost podatkovne strukture. Zatem pa bodo predstavljene implementacije poizvedb nad vhodno besedo s pomočjo kompaktne predstavitve priponskega drevesa.

V šestem poglavju bo izdelana primerjava med različnimi indeksi, ki so bili predhodno predstavljeni. Izdelana bo empirična primerjava različnih indeksov, ki bo merila prostorsko zahtevnost podatkovnih struktur in časovno zahtevnost izgradnje ter poizvedb nad njimi.

## 2 OZNAKE, DEFINICIJE IN OSNOVNE OPERACIJE

V tem poglavju bodo predstavljene osnovne oznake in definicije, ki bodo uporabljene skozi celotno magistrsko nalogo. Zatem pa bodo predstavljene osnovne podatkovne strukture in predstavitve podatkovnih struktur, ki so potrebne za implementacijo kompaktne predstavitve priponskih dreves.

### 2.1 OZNAKE IN OSNOVNE DEFINICIJE

V celotni magistrski nalogi bo  $\Sigma$  predstavljala abecedo. Abeceda je množica znakov  $\Sigma = \{\sigma_1, \dots, \sigma_a\}$ . Znake v  $\Sigma$  lahko uredimo v zaporedje ter definiramo, da je  $\sigma_i$  manjši od  $\sigma_j$ , ki je različen od  $\sigma_i$ , natanko tedaj, ko se znak  $\sigma_i$  pojavi pred znakom  $\sigma_j$  v zaporedju. Sicer je  $\sigma_i$  večji od  $\sigma_j$ . Vhodna beseda  $T$  dolžine  $n$  znakov, iz katere bomo zgradili priponsko drevo, priponsko polje ali kompaktno priponsko drevo, je sestavljena iz znakov abecede  $\Sigma$ , torej je  $T \in \Sigma^n$ . Na podoben način  $P$  predstavlja iskani vzorec dolžine  $m$  znakov, za katerega želimo preveriti prisotnost v  $T$ . Tako kot beseda  $T$  je tudi vzorec  $P$  sestavljen iz znakov abecede  $\Sigma$ , torej je  $P \in \Sigma^m$ .

Niz znakov iz abecede  $\Sigma$ , ki je del besede in se začne na  $i$ -tem in se konča na  $j$ -tem znaku besede, imenujemo podniz in ga označimo kot  $T[i : j]$ . Posebna vrsta podniza, ki se začne na  $i$ -tem znaku in konča na koncu besedila, imenujemo pripona ter se jo označi kot  $T[i : n]$  oziroma  $T[i : \cdot]$ . Poljubni nizi oziroma podnizi so označeni z grškimi črkami  $\alpha$ ,  $\beta$  in  $\gamma$ , pri čemer je navedeno, ali gre za podniz ali niz. Formalno je podniz definiran na sledeč način:

**Definicija 2.1.** Niz  $\alpha \in \Sigma^*$  je podniz besede  $T$ , ko obstaja tak  $1 \leq i \leq n$ , za katerega velja, da je  $\alpha = T[i : i + |\alpha|]$ .

Dva poljubna niza se lahko združita v nov daljši niz s stikom, ki je definiran na sledeči način:

**Definicija 2.2.** Stik nizov  $\alpha$  in  $\beta$  je niz  $\gamma = \alpha \cdot \beta = \alpha\beta$ , pri čemer je  $\alpha = \gamma[1 : |\alpha|]$  in  $\beta = \gamma[|\alpha| + 1 : |\alpha| + |\beta|]$ .

Za razliko od nizov in podnizov so znaki iz abecede  $\Sigma$  označeni s črkama  $x$  ali  $t$ . Oznaka  $T[i]$  pa predstavlja znak na  $i$ -tem mestu vhodne besede.

Ker bo nad besedo  $T$  izgrajeno priponsko drevo ali kompaktno priponsko drevo, ki je uporabljeno za hitrejše iskanje vzorcev v besedi, so skozi nalogo vozlišča označena s črkami  $s$ ,  $v$ ,  $w$  in  $u$ . Listi priponskega drevesa pa bodo označeni s črko  $l$ , kadar govorimo splošno o listih, sicer pa bodo označeni z  $l_i$ , pri čemer  $i$  označuje zaporedno število lista od leve proti desni.

Skozi nalogo bodo  $\log_2 n$  označeni kot  $\log n$ . Če zapisan logaritem ni dvojiški logaritem, bo osnova tega označena.

## 2.2 ENIŠKI ZAPIS

Kot je iz samega imena zapisa razvidno, bodo števila predstavljena v eniškem sistemu. Torej bo nenegativno celo število  $s$  predstavljeno kot  $1^s0$ , pri čemer je abeceda  $\Sigma = \{0, 1\}$ , s  $s$ -timi enicami, medtem ko so števila med seboj ločena z ničlo. Na primer, seznam števil  $[1, 3, 0, 4, 1]$  bi bil v eniškem zapisu predstavljen kot niz  $10111001111010$ .

Eniški zapis bo uporabljen pri kompaktni predstavitvi dreves. Razlog za uporabo eniškega zapisa števil namesto dvojiškega je lažja pretvorba operacij nad drevesi v operacije nad bitnimi polji, saj imajo na ta način enice in ničle vsaka svoj pomen. Ker je zapis uporabljen za predstavitev stopenj vozlišč, zaporedje oblike  $1^s0$  predstavlja eno vozlišče. Vsaka enica v vozlišču predstavlja otroka vozišča, ničla pa predstavlja, da je bilo vozlišče že obiskano v zapisu.

## 2.3 MODEL RAČUNANJA

Preden se lahko predstavi način shranjevanja bitnega polja, je najprej potrebno predstaviti uporabljen model računanja. Uporabljen bo splošen model računanja, saj se različne arhitekture mikroprocesorjev razlikujejo med seboj in bi bilo potrebno narediti analizo za vsako arhitekturo posebej. Zato bo izbran model računanja računalnik z naključnim dostopom (angl. *random-access machine* oziroma RAM). Natančneje, uporabljen bo besedni RAM (angl. *word RAM*).

Osnovna različica RAM modela predpostavlja, da so vse aritmetične operacije nad celimi števili in logične operacijami nad biti izvedene v konstantnem času. Poleg tega RAM omogoča dostop do pomnilnika v konstantnem času ter linearen čas za dodelitev zaporednega spomina. Vse te operacije so izvedene v enakem času tudi v besednem RAM modelu. Edina razlika med modeloma je, da RAM predpostavlja neskončno velik pomnilnik, medtem ko besedni RAM omeji velikost pomnilnika na  $U$  naslovov. Posledično je velikost ene pomnilniške besede  $w = \log U$  bitov, kar je velikost enega naslova [6, 22, 23]. Pri tem se lahko definira tudi »cela števila« (angl. *Integer*). »Cela števila« so definirana kot podmnožica celih števil, ki so lahko predstavljena v dvojiškem

zapisu z eno računalniško besedo [6].

## 2.4 BITNO POLJE

Bitno polje (angl. *bit array*)  $B$  dolžine  $b = |B|$  je polje, v katerem je shranjenih  $b$  bitov. Ker je dostop do posameznega bita možen v konstantnem času, se bitno polje lahko uporabi kot osnovna podatkovna struktura za implementacijo kompaktne predstavitve drugih podatkovnih struktur. Primer take podatkovne strukture, ki uporablja bitno polje za svojo kompaktno predstavitev, je drevo.

V nadaljevanju tega podpoglavja bo predstavljeno, kako učinkovito shraniti bitno polje ter ohraniti dostop do podatkov v konstantnem času. Predstavljene bodo še druge operacije nad bitnimi polji, ki se uporabljajo za reševanje problemov, predstavljenih z bitnimi polji. Predstavljene operacije so:  $dostop(B, i)$  (angl. *access*),  $rang(B, i)$  (angl. *rank*),  $izbira(B, i)$  (angl. *select*),  $predhodnik(B, i)$ ,  $NaslednikBi$  in dodatne operacije nad območji (angl. *range query*) bitnega polja. Vse predstavljene operacije želimo opraviti čim bolj učinkovito ter pri tem uporabiti čim manj dodatnega prostora.

**Shramba in dostop.** Intuitivni način shranjevanja bitnega polja  $B$  dolžine  $b$  v spominu bi bil tak, da bi se vsaka celica polja shranila v pomnilniku na svojem naslovu. Na ta način bo operacija  $dostop(B, i)$  oziroma dostop do celice  $B[i]$  potreboval konstanten čas, da se izvede. Pri tem pa bo bitno polje potrebovalo  $O(b)$  bitov oziroma  $wb$  bitov. Na ta način ima vsaka celica bitnega polja  $w - 1$  odvečnih bitov.

Zato se pojavi vprašanje, ali se je mogoče izogniti teh  $b(w - 1)$  odvečnih bitov. Če se je mogoče izogniti uporabi teh odvečnih bitov, bi bitno polje  $B$  potrebovalo zgolj  $b + o(b)$  bitov (dodatnih  $o(b)$  bitov je potrebnih, če  $b$  ni večkratnik od števila  $w$ ). Torej bi se lahko bitno polje  $B$  predstavilo kot polje pomnilniških besed  $W \left[1 : \left\lceil \frac{b}{w} \right\rceil\right]$ . Naslednji problem, ki ga je potrebno rešiti, je, kako učinkovito dostopati do  $i$ -tega bita v bitnem polju. V konstantnem času lahko dostopamo do celice v polju  $W$ , v kateri je shranjen  $i$ -ti bit. Ta je shranjen v celici  $W \left[\left\lceil \frac{i}{w} \right\rceil\right] = w_i$ . Naivna metoda dostopa do bita v pomnilniški besedi bi bila z bitnim premikom (angl. *bit shift*) v desno, ponovljenim  $(w - r)$ -krat, pri čemer je  $r = ((i - 1) \bmod w) + 1$ . Za to je potrebno  $O(w)$  časa. Ker pa je en premik v desno enakovreden celoštevilskemu deljenju števila z dva, se lahko nadomesti premike v desno s celoštevilskemu deljenjem števila, ki ga predstavlja pomnilniška beseda  $w_i$ , s številom  $2^{w-r}$ . Torej vrednost bita  $B[i]$  je izračunana kot:

$$\left\lfloor W \left[ \left\lceil \frac{i}{w} \right\rceil \right] / 2^{w-r} \right\rfloor \bmod 2.$$

Ker besedni RAM model predpostavi, da so vse aritmetične operacije nad celimi števili, med katere sodi tudi celoštevilsko deljenje, opravljene v konstantnem času, je čas potre-

ben za dostop do  $i$ -tega elementa tudi konstanten. Torej se je mogoče izogniti uporabi odvečnih bitov iz naivne implementacije, ne da bi se časovna zahtevnost povečala [6].

**Rang.** Prva operacija, ki jo definiramo nad bitnim poljem  $B$  velikosti  $b$ , je  $\text{rang}_v(B, i)$ . Operacija vrne število bitov z vrednostjo  $v \in \{1, 0\}$  v  $B$  do vključno položaja  $i$ . V nadaljevanju bo operacija  $\text{rang}(B, i)$  predstavljala  $\text{rang}_1(B, i)$ . To je možno, saj velja sledeča zveza med  $\text{rang}_0(B, i)$  in  $\text{rang}_1(B, i)$ :

$$\text{rang}_1(B, i) = i - \text{rang}_0(B, i). \quad (2.1)$$

Operacijo je možno naivno implementirati s štetjem bitov z vrednostjo 1 v bitnem polju, za kar je potrebno  $O(b)$  časa. Šetje bitov z vrednostjo 1 se v angl. imenuje *population count* (v nadaljevanju označeno **popcount**), ki se izračuna v konstantnem času za eno pomnilniško besedo [27, 29]. Zaradi uporabnosti te funkcije je ta že implementirana v različnih modernih arhitekturah procesorjev in je zato možno uporabiti strojno operacijo. Operacijo rang se lahko pospeši na konstantni čas, pri tem pa se potrebujejo dodatne podatkovne strukture, ki zasedejo  $o(b)$  bitov. Pri tem ni potrebno zgraditi dodatnih podatkovnih struktur za obe različici ranga, saj relacija iz enakosti (2.1) omogoča izgradnjo podatkovne strukture zgolj za operacijo  $\text{rang}_1$ .

Pomožna struktura shranjuje range različnih elementov bitnega polja  $B$  v polju  $R$ . Naivni pristop bi shranil v polju  $R$  range vseh elementov  $B$ -ja. Problem tega pristopa je prevelika prostorska zahtevnost, saj je zaželeno, da je  $R$  čim manjši in po možnosti ni večji od  $o(b)$  bitov. Predlagana rešitev potrebuje  $b \log r_1 = O(bw)$  dodatnih bitov, pri čemer je  $r_1$  število bitov z vrednostjo 1 v bitnem polju  $B$ .

Rešitev tega problema je vzorčenje rangov, tako da so v polju  $R$  shranjeni zgolj rangi nekaterih elementov  $B$ -ja. Polje  $R$  razdeli  $B$  na  $s = kw$  delov, pri čemer je  $k$  poljubno število. Element  $R[i] = \text{rang}_1(B, is)$ , pri čemer je  $0 \leq i \leq \lfloor \frac{b}{s} \rfloor$ . Na ta način je operacija rang implementirana kot

$$\text{rang}_1(B, i) = R \left[ \left\lfloor \frac{i}{s} \right\rfloor \right] + \text{popcount} \left( B \left[ \left\lfloor \frac{i}{s} \right\rfloor, i \right] \right).$$

Polje  $R$  ima velikost  $\lfloor \frac{b}{k} \rfloor$  bitov, saj shranjuje  $\lfloor \frac{b}{s} \rfloor$  števil velikosti  $w$  bitov. Pri tem je potrebno funkcijo **popcount** izvesti največ  $k$ -krat, kar naredi čas operacije rang  $O(k)$  [6]. Na Sliki 1 je prikazan primer polja  $R$  za podano bitno polje  $B$ .

Podobno kot polje  $R$  se definira tudi polje  $R'$ . Polje  $R'$  hrani na  $i$ -tem mestu število bitov z vrednostjo 1 po vedru  $R[\lfloor \frac{i}{k} \rfloor]$  ali  $R'[i] = \text{rang}_1(B, iw) - R[\lfloor \frac{i}{k} \rfloor]$ . Na ta način se zniža število klicev funkcije **popcount** na 1 klic. Ker so v  $R'$  shranjena zgolj števila med 0 in  $s$  (vsako vedro v  $R$  ima  $k$  elementov v  $R'$ ), se lahko  $R'$  shrani v  $\lfloor \frac{b}{w} \rfloor \log s$  bitov, kar je  $o(b)$  bitov. Tako je operacija rang implementirana z uporabo polj  $R$  in  $R'$ . In sicer je implementirana na sledeči način:

$$\text{rang}_1(B, i) = R \left[ \left\lfloor \frac{i}{kw} \right\rfloor \right] + R' \left[ \left\lfloor \frac{i}{w} \right\rfloor \right] + \text{popcount} \left( B \left[ \left\lfloor \frac{i}{w} \right\rfloor, \left\lfloor \frac{i}{w} \right\rfloor + (i \bmod w) \right] \right).$$



$R'$	0	3	5	8	0	1	2	4	0
$R$	1				9				16
$B$	1011	1100	1101	1000	0000	1001	0110	0101	1011

Slika 1: Primer dodatne podatkovne strukture za operacijo  $\text{rang}_1$  v konstantnem času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine  $w = 4$ . Pri tem se je izbralo, da je število  $k = 4$ .

Ta implementacija potrebuje konstanten čas za izračunati  $\text{rang}(B, i)$ . Dostop do polj  $R$  in  $R'$  je storjen v konstantnem času, prav tako pa tudi izračun funkcije `popcount`, ki je klicana samo enkrat [6]. Na Sliki 1 je prikazano tudi polje  $R'$ .

**Izbira:** Druga osnovna operacija nad bitnim poljem  $B$  je  $\text{izbira}_v(B, i)$ , ki vrne položaj  $i$ -te ponovitve vrednosti  $v \in \{1, 0\}$  v  $B$ . Podobno kot operacija  $\text{rang}$ , tudi operacija izbira potrebuje pomožno podatkovno strukturo za izvedbo v konstantnem času.

Operacijo izbira si lahko predstavimo kot inverzno operacijo operacije  $\text{rang}$ , saj velja zveza  $j = \text{rang}_v(B, \text{izbira}_v(B, i))$ . Pri tem pa ne obstaja povezava med operacijo  $\text{izbira}_1(B, i)$  in  $\text{izbira}_0(B, i)$ . To pomeni, da rešitev s pomožno podatkovno strukturo za  $\text{izbira}_1(B, i)$  ne more biti uporabljena za iskanje rešitve za  $\text{izbira}_0(B, i)$ . V nadaljevanju bo opisan časovno učinkovit postopek iskanja  $\text{izbira}_1(B, i)$ , saj se lahko  $\text{izbira}_0(B, i)$  implementira na podoben način [6].

Ko operacija izbira ni ključna pri reševanju problemov, se lahko uporabi binarno iskanje v poljih  $R$  in  $R'$ , ki sta del podatkovne strukture za  $\text{rang}$ , za kar je potrebno  $O(\log b)$  časa. Z binarnim iskanjem se najde območje dolžine  $k$ , pri čemer je potrebno še dodatnih  $k$  preiskav, da se najde natančno vrednost. Čas operacije se lahko zniža na  $O(\log \log b)$  z uporabo pomožnih podatkovnih struktur. Podobno kot pri operaciji  $\text{rang}$  se bitno polje razdeli na  $\lceil \frac{r_1}{s} \rceil$  veder, pri čemer pa je  $r_1$  število bitov z vrednostjo 1 v bitnem polju in  $s$  je število takih bitov v vsakem vedru. Polje  $S[0 : \lceil \frac{r_1}{s} \rceil]$  hrani vrednosti  $S[i] = \text{izbira}_1(B, i \cdot s + 1)$ , pri čemer  $S[\lceil \frac{r_1}{s} \rceil] = b + 1$ . Polje  $S$  potrebuje  $w(\lceil \frac{r_1}{s} \rceil + 1)$  bitov. Ko je  $s = w^2$ , potem podatkovna struktura potrebuje  $w(\lceil \frac{r_1}{w^2} \rceil + 1) = o(r_1) = o(b)$  bitov [6].

Vedra niso enako velika, zato se lahko razdelijo na velika vedra in mala vedra, pri čemer je vedro veliko natanko tedaj, ko je večje od  $s \log^2 b$  bitov, sicer je malo vedro. Pri tem je potrebno shraniti velikost vedra v bitno polje  $V$ , kjer  $V[i] = 1$ , če  $i$ -to vedro je veliko, sicer je  $V[i] = 0$ . Za velika vedra se izračunajo vseh  $s$  vrednosti izbire, pri čemer so shranjeni v polju  $I$  položaji bitov z vrednostjo ena znotraj vedra. Za mala

$$izbira_1(B, j) = \begin{cases} I \left[ (rang_1(V, \left\lceil \frac{j}{s} \right\rceil) - 1)s + x \right], & V \left[ \left\lceil \frac{j}{s} \right\rceil \right] = 1 \\ S \left[ \left\lceil \frac{j}{s} \right\rceil - 1 \right] + k, & V \left[ \left\lceil \frac{j}{s} \right\rceil \right] = 0 \\ b + 1, & r_1 < j \end{cases},$$

				$I$	13	21	24	26				
$S$	1	6	13	27	35	37	$V$	0	0	1	0	0
$B$	1011	1100	1101		1000	0000		1001	0110	0101	1011	

Polje  $I$  mora shraniti  $s \lceil \log b \rceil$  bitov za vsako veliko vedro, ki jih je  $\frac{b}{s(\log b)^2}$ , torej potrebuje  $O\left(\left\lceil \frac{b}{\log b} \right\rceil\right) = o(b)$  bitov. Bitno polje  $V$  pa potrebuje  $\lceil \frac{m}{s} \rceil$  bitov za shraniti velikosti blokov in dodatne bite za izvajanje operacije rang v konstantnem času, torej tudi  $V$  potrebuje  $o(b)$  bitov [6].

Vse dodatne podatkovne strukture, ki so potrebne za izvajanje operacij rang in izbira v konstantnem času, so lahko izgrajene v dveh sprehodih po bitnem polju  $B$ , pri čemer vsak sprehod traja  $O(b)$  časa. Pri tem je bil ves potreben spomin že predhodno dodeljen.

**Predhodnik/Naslednik.** S pomočjo operaciji rang in izbira lahko implementiramo dodatne operacije, ki omogočajo lažje iskanje po bitnem polju  $B$ . Dve taki operaciji, ki

bosta uporabljeni za implementacijo operacij nad drevesi, sta predhodnik in naslednik.

Operacija predhodnik elementa  $y$  najde največji indeks  $i < y$ , za katerega velja, da je  $B[i] = v$ . Ta operacija je implementirana kot

$$\text{predhodnik}_v(B, y) = \text{izbira}_v(B, \text{rang}_v(B, y)),$$

pri čemer je  $v \in \{0, 1\}$ . Na podoben način se definira tudi operacijo naslednik. Operacija naslednik elementa  $y$  najde najmanjši indeks  $i > y$ , za katerega velja, da je  $B[i] = v$ . Pri tem je operacija implementirana kot

$$\text{naslednik}_v(B, y) = \text{izbira}_v(B, \text{rang}_v(B, y - 1) + 1),$$

kjer je  $v \in \{1, 0\}$  [6].

Operaciji se uporabljata za sprehod po bitnem polju  $B$ . Z njima se lahko najde indekse vseh bitov z vrednostjo  $v$  v  $O(r_v)$  časa ( $r_v$  je število bitov z vrednostjo  $v$  v bitnem polju  $B$ ).

**Višek.** Do sedaj so vse predstavljene operacije imele kot vhod zgolj en element v bitnem polju, ampak nekateri problemi zahtevajo rešitev za podani interval. Pogosta vprašanja na intervalih sta iskanje največjega ali najmanjšega števila v intervalu. Na bitnih poljih pa se ta problem pretvori v razliko med številom bitov z vrednostjo 0 ter številom bitov z vrednostjo 1 do  $i$ -tega bita. To razmerje imenujemo  $\text{višek}(B, i)$  in je izračunan kot:

$$\text{višek}(B, i) = \text{rang}_0(B, i) - \text{rang}_1(B, i) = 2 \text{rang}_0(B, i) - i.$$

Na podoben način se lahko definira tudi relativni višek na intervalu med indeksom  $i$  in  $j$ , in sicer kot:

$$\text{višek}(B, i, j) = \text{višek}(B, j) - \text{višek}(B, i - 1).$$

Višek in posledično tudi relativni višek sta izračunana zgolj z uporabo ranga, zato jih je možno izračunati v konstantnem času.

**Operacije na območjih.** Poznamo različne operacije na območjih, ampak za potrebe magistrske naloge bomo definirali štiri operacije. Prva operacija je  $\text{rmq}(B, i, j)$  (angl. *range minimum query*), ki vrne indeks  $k$ , za katerega velja, da je  $i \leq k \leq j$  in  $\text{višek}(B, k)$  je najnižji višek na intervalu med  $i$  in  $j$  ter se najnižji višek prvič pojavi na indeksu  $k$ . Podobno je definirana tudi operacija  $\text{rMq}(B, i, j)$  (angl. *range maximum query*), ki pa vrne indeks  $k$ , pri čemer je  $i \leq k \leq j$  in  $\text{višek}(B, k)$  je najvišji višek na intervalu med  $i$  in  $j$  ter se najvišji višek prvič pojavi na indeksu  $k$ . Operacija  $\text{minIzbira}(B, i, j, t)$  vrne položaj  $t$ -tega pojava najmanjšega viška na intervalu med  $i$  in  $j$ . Operacija  $\text{minŠtetje}(B, i, j)$  pa vrne število pojav najnižjega viška na intervalu

med  $i$  in  $j$ . S pomočjo teh operaciji bodo v nadaljevanju implementirane operacije nad drevesi v enem od kompaktnih prikazov [6].

Vse predstavljene operacije na intervalih so lahko implementirane s časovno zahtevnostjo  $O(\log b)$ . Pri tem pa je potrebno zgraditi dodatno podatkovno strukturo *rmM*-drevo (angl. *range minimum maximum tree*), ki je lahko shranjeno z  $O(b/\log b) = o(b)$  dodatnimi biti. Drevo je levo poravnano in se lahko zapiše kot polje vozlišč (podobno kot podatkovna struktura kopica). Torej so otroci  $i$ -tega vozlišča na  $2i$ -tem in  $(2i+1)$ -vem mestu v polju ter starš  $i$ -tega vozlišča se nahaja na  $\lfloor \frac{i}{2} \rfloor$ -mestu. Vsako vozlišče drevesa predstavlja interval v bitnem polju. Prvi otrok vozlišča pokriva prvo polovico intervala, drugi otrok pa pokriva drugo polovico intervala. Torej koren *rmM*-drevesa pokriva celotno bitno polje  $B$ . Ker želimo časovno in prostorsko učinkovito podatkovno strukturo lahko  $B$  razdelimo na na  $\frac{b}{a}$  veder velikosti  $a$  elementov. Posledično lahko vsak interval dolžine  $a$  predstavlja list *rmM*-drevesa. Za zagotoviti prostorsko zahtevnost  $o(b)$  bitov in časovno zahtevnost  $O(\log b)$  izberemo vrednost  $a = \log b$ . Vsako vozlišče v drevesu hrani štiri podatke:  $e$  relativni višek pokritega intervala,  $m$  najmanjši relativni višek v območju,  $M$  največji relativni višek na območju in  $n$  število najmanjših viškov na pokritem intervalu. Pri tem vozlišče pokriva celotno območje, ki ga pokrivata oba otroka, in listi pokrivajo zgolj eno vedro velikosti  $a$  elementov. Torej koren drevesa pokriva celotno bitno polje  $B$ . Vse predstavljene operacije so implementirane s pomočjo sprehoda po *rmM*-drevesu [6].

Vse štiri operacije temeljijo na sprehodu po *rmM*-drevesu, torej bo sprehod predstavljen zgolj za operacijo  $rmq(B, i, j)$  na intervalu bitnega polja  $B[i : j]$ . Operacija  $rmq(B, i, j)$  je izvedena v dveh korakih. Prvi korak je iskanje vrednosti najmanjšega viška na intervalu  $B[i : j]$ . Ker se lahko  $i$  nahaja sredi vedra bitov, se izračunata relativni višek  $d$  in najnižji višek  $m$  na intervalu med  $i$  in začetkom vedra  $\lceil i/a \rceil$  brez uporabe *rmM*-drevesa. Če je  $j$  med  $i$  in začetkom vedra  $\lceil i/a \rceil$ , se izračuna zgolj do  $j$ -tega bita in  $m$  predstavlja tudi najnižji višek na intervalu med  $i$  in  $j$ , za kar potrebujemo  $O(a)$  časa. Sicer pa se nadaljuje sprehod po *rmM*-drevesu. Začetno vozlišče sprehoda je list, ki predstavlja vedro  $\lceil i/a \rceil$ , in ga označimo z  $v$ . Izračunamo tudi zaporedno število lista, ki vsebuje vedro s koncem intervala  $j$ , in ga označimo kot  $k$ . Zdaj lahko začnemo s sprehodom po drevesu navzgor. Če je  $v$  desni otrok ( $v$  je sodo število), se premaknemo v starša, torej  $v \leftarrow (v-1)/2$ . Sicer pa je potrebno preveriti, ali je desni brat tudi v intervalu, kar se lahko preveri tako, da preverimo, če trditev  $\lfloor l/2^{\lfloor \log l \rfloor - \lfloor \log u \rfloor} \rfloor \neq u$  drži, potem je vozlišče  $u$  tudi znotraj intervala  $B[i : j]$ . Če je desni brat v intervalu preverimo, ali smo našli nov najmanjši višek na intervalu ( $m > d + rmM[v+1].m$ ). Če smo ga našli, potem  $m \leftarrow d + rmM[v+1].m$ . Nato pa se še popravi višek intervala na  $d \leftarrow d + rmM[v+1].e$ . Zatam pa se premaknemo v starša, torej  $v \leftarrow v/2$  [6].

Ko desni brat ni v intervalu, torej ne velja prejšnji pogoj, potem nadaljujemo s sprehodom navzdol iz našega desnega brata  $v \leftarrow (v+1)/2$ . Tokrat pa preverjamo, ali je levi

otrok vsebovan v intervalu in ali vozlišče  $v$  omogoča zmanjševanje najmanjšega viška. Če vozlišče  $v$  ne omogoča zmanjševanja, lahko sprehod končamo in vemo, da je najmanjši višek  $m$ , sicer pa nadaljujemo s sprehodom. Če levi otrok ni vsebovan, nadaljujemo s sprehodom v levem otroku, sicer pa preverimo, če velja  $m > d + rmM[2v].m$  in v primeru, da velja popravimo vrednost  $m$ . Nato popravimo vrednost  $d \leftarrow d + rmM[2v].e$  in nadaljujemo z iskanjem v desnem otroku  $v \leftarrow 2v + 1$ . Sprehod nadaljujemo, dokler ne dosežemo listov oziroma dokler je  $v > \lceil n/a \rceil$ . Če list  $v$  ne omogoča zmanjševanja najmanjše vrednosti viška na intervalu oziroma  $m \leq d + rmM[v].m$ , potem je  $m$  najmanjši višek na intervalu. Sicer pa pregledamo, na podoben način kot pred začetkom iskanja, še višek do  $j$ -tega bita in najmanjši višek na tem intervalu označimo  $m'$ . Če  $m > d + m'$ , potem je najmanjši višek na intervalu  $B[i : j]$  enak  $d + m'$ , sicer pa je  $m$  [6].

Zadnji korak pa je iskanje točnega indeksa bita z relativnim viškom  $m$ . To je storjeno z operacijo *iskanjeNaprej*( $B, i, m$ ), ki najde prvi  $j > i$ , za katerega velja  $višek(B, j) = višek(B, i) + m$ . Operacija je implementirana na isti način, samo tokrat se ne preverja vrednost polja  $rmM[v].m$ , ampak se išče prvo pojavitev viška, ki je enaka  $m$ . Ta isti postopek se lahko uporabi tudi za preostale tri operacije, pri čemer pa operacija  $rmq(B, i, j)$  uporablja namesto polja  $rmM[v].m$  polje  $rmM[v].M$  in relacija manjše postane večje in obratno. Operaciji *minIzbira*( $B, i, j, t$ ) in *minŠtetje*( $B, i, j$ ) pa uporabljata polje  $rmM[v].n$  [6].

Tako implementirane operacije potrebujejo  $O(\log b)$  časa, da se izvršijo. Začetno in končno štetje bitov potrebuje  $O(a) = O(1)$  čas. Ker je  $rmM$ -drevo dvojiško in polno (vsaka globina drevesa, razen zadnje, ima maksimalno število otrok) drevo, je njegova višina enaka  $O(\log \lceil b/a \rceil) = O(\log b)$ , torej tudi sprehodi, ki gredo od lista proti korenu ali od korena proti listu, potrebujejo  $O(\log b)$  časa. Z izdelavo manjših  $rmM$ -dreves, ki zavzamejo  $\beta = \log^3 b$  elementov, in dodatne podatkovne strukture imenovane pospeševalnik (angl. *accelerator*), je možno implementirati te operacije v času  $O(\log \log b)$  [6].

## 2.5 KOMPAKTNA PREDSTAVITEV DREVES

Z uporabo bitnih polj je mogoče predstaviti mnogo podatkovnih struktur. Ker pa se magistrska naloga osredotoča na priponska drevesa, bo zato predstavljena uporaba bitnih polj za kompaktno predstavitev topologije dreves. Običajno je podatkovna struktura drevo sestavljena iz vozlišč ter referenc na njih. V vsakem vozlišču je shranjena vrednost, ki jo predstavlja vozlišče, ter referenca na svoje otroke. Drevo z  $n$ -timi vozlišči potrebuje  $O(n \log n)$  bitov za shraniti topologijo drevesa (v praksi potrebuje  $O(nw)$  bitov). Kompaktna predstavitev topologije dreves zniža prostorsko zahtevnost

na  $2n + o(n)$  bitov. Za vsako vozlišče je potrebno predstaviti referenco na vozlišče ter predstaviti, da je vozlišče bilo že obiskano, zato vsako vozlišče potrebuje 2 bita oziroma za predstavitev celotnega drevesa je potrebnih  $2n$  bitov.

Pri tem obstajajo tri vrste kompaktne predstavitve dreves: Uravnoteženi oklepaji (angl. *Balanced Parentheses* oziroma BP), Zaporedje eniških zapisov stopenj vozlišč po plasteh (angl. *Level Order Unary Degree Sequence* oziroma LOUDS) in Zaporedje eniških zapisov stopenj vozlišč v globino (angl. *Depth-First Unary Degree Sequence* oziroma DFUDS). Naslednja definicija prikaže operacije nad drevesi.

**Definicija 2.3.** Podatkovne struktura drevo mora podpirati naslednje operacije:

1.  $koren()$  vrne koren drevesa,
2.  $jeList(v)$  vrne *true*, če je vozlišče  $v$  list, sicer pa vrne *false*,
3.  $stOtrok(v)$  vrne število otrok vozlišča  $v$ ,
4.  $otrok(v, i)$  vrne vozlišče  $w$ , ki je  $i$ -ti otrok vozlišča  $v$ ,
5.  $prviOtrok(v)$  vrne vozlišče  $w$ , ki je prvi otrok vozlišča  $v$ ,
6.  $nBrat(v)$  vrne vozlišče  $w$ , ki je desni (naslednji) brat od vozlišča  $v$ ,
7.  $pBrat(v)$  vrne vozlišče  $w$ , ki je levi (predhodni) brat od vozlišča  $v$ ,
8.  $starš(v)$  vrne vozlišče  $w$ , ki je starš od vozlišča  $v$ ,
9.  $globina(v)$  vrne število vozlišč na poti iz korena do vozlišča  $v$ ,
10.  $lca(v, w)$  vrne najnižjega skupnega prednika od  $v$  in  $w$ .

Iz definicije se lahko opazi, da večina operacij mogoče implementirati zgolj z operacijo  $Otrok(v, i)$  ter operacijo  $starš(v)$ , s katerimi se lahko sprehajamo po drevesu. S pomočjo operacij iz definicije je možno implementirati ostale operacije na drevesih, ki so bolj specifične za posamično implementacijo drevesa, na primer *vstavi*, *izbriši*, *najmanjši\_element* (v kopici) in ostale.

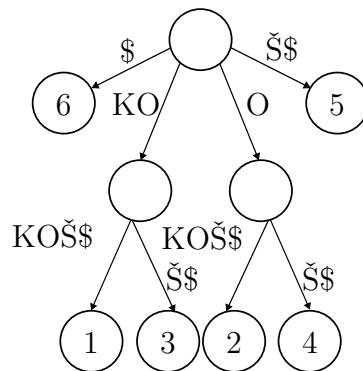
### 2.5.1 Zaporedje eniških zapisov stopenj vozlišč po plasteh

Prvi način zapisa topologije drevesa je zaporedje eniških zapisov stopenj vozlišč po plasteh (LOUDS). Ideja zapisa temelji na sprehodu po plasteh po drevesu in zapisu stopenj vozlišč ob njihovem obisku. Pri tem pa se pojavi problem, kako implementirati sprehod po polju stopenj in posledično po drevesu. Potrebni sta dodatni strukturi, in sicer za vsoto stopenj vozlišč do trenutnega vozlišča ter strukturo za štetje vozlišč. Obe strukturi lahko implementiramo kot polji števil. Z uporabo eniškega zapisa, pa

se te dve strukturi prevedeta na operaciji *rang* in *izbira* nad bitnim poljem z eniškim zapisom.

Topologijo drevesa se predstavi kot bitno polje  $B$  dolžine  $2n + 1$ , pri čemer je  $n$  število vozlišč v drevesu. Zapis drevesa se začne z nizom 10, temu pa sledijo stopnje vsakega vozlišča v eniškem zapisu. Kot je razvidno iz imena predstavitve, so vozlišča obiskana po nivojih: vozlišča z isto globino so zaporedno predstavljena [6].

Primer takega zapisa je predstavljen na Sliki 3. Na sliki so vozlišča v zaporedju ločena s sivimi črtami. Koren drevesa je predstavljen z  $B[3 : 7] = 11110$ . Niz  $B[3 : 7]$  vsebuje 4 bite z vrednostjo 1, ker ima koren 4 otroke.



10|11110|0|110|110|0|0|0|0|0

Slika 3: Primer predstavitve drevesa z metodo LOUDS za priponsko drevo besede »KOKOŠ\$«.

Tabela 1: Implementacija operacij drevesa v LOUDS.

Operacija	Implementacija v LOUDS
$koren()$	3
$jeList(v)$	$B[v] == 0$
$stOtrok(v)$	$naslednik_0(B, v) - v$
$otrok(v, i)$	$izbira_0(B, rang_1(B, v - 1 + 1)) + 1$
$prviOtrok(v)$	$otrok(v, 1)$
$nBrat(v)$	$naslednik_0(B, v) + 1$
$pBrat(v)$	$predhodnik_0(B, v - 2) + 1$
$starš(v)$	$predhodnik_v(B, izbira_1(B, rang_0(B, v - 1))) + 1$
$globina(v)$	/
$lca(v, w)$	Algoritem 1

Med izgradnjo kompaktne predstavitve se vozlišča indeksira s števili  $i$  med 1 in  $n$ . Število  $i$  predstavlja zaporedno število obiskanega vozlišča, torej koren ima vrednost

1, prvi otrok korena ima vrednost 2 in tako dalje. Indeks se uporabi za shranjevanje vrednosti, ki so po navadi shranjene znotraj vozlišča. V Tabeli 1 so predstavljene implementacije operacij z uporabo predstavitve drevesa LOUDS. Indek vozlišča se izračuna kot  $izbira_1(B, v) + 1$  [6].

Vse operacije razen operacije *globina* in *lca* so izvršene v konstantnem času. Pri tem je treba izgraditi podatkovno strukturo zgolj za  $izbira_0$ , saj operacija  $izbira_1$  ni potrebna za pravilno delovanje operacij drevesa. Operacija *globina* ni podprta, saj LOUDS predstavitev ne omogoča učinkovitega iskanja. Operacija *globina* je lahko implementirana s štetjem, koliko krat se uporabi operacija  $stars(v)$  za doseči *koren* drevesa. Pri tem pa velja, da je  $globina(u) \geq globina(v)$ , ko je  $u > v$ . S pomočjo tega dejstva se lahko implementira operacijo *lca*, kot je prikazano v Algoritmu 1 [6].

---

**Algoritem 1:** Operacija  $lca(v, w)$  (LOUDS)

---

**Vhod:** Bitno polje  $B$ , vozlišča  $v$  in  $w$

**Izhod:** Vozlišče  $u$

```

1 while  $v \neq w$  do
2   če  $v > w$  potem
3      $v \leftarrow stars(v)$ 
4   sicer
5      $w \leftarrow stars(w)$ 
6 vrni  $v$ 

```

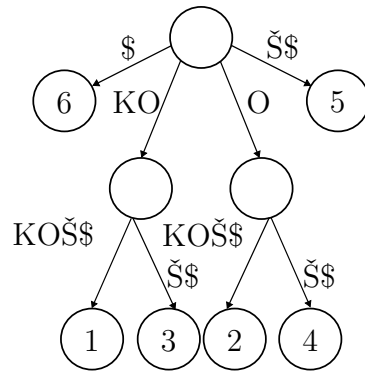
---

### 2.5.2 Zaporedje eniških zapisov stopenj vozlišč v globino

Naslednja predstavitev topologije drevesa je zaporedje eniških zapisov stopenj vozlišč v globino (DFUDS). Ideja predstavitve temelji na premem sprehodu (angl. *Preorder traversal*) po drevesu. Ko prvič obiščemo, zapišemo njegovo stopnjo. Pri tem lahko opazimo, da so poddrevesa otrokov vozlišča  $v$  zapisana v zaporednih celicah polja. Posledično se poddrevo naslednja brat od  $v$  začne po skrajno desnem listu v poddrevesu s korenov v  $v$ . Torej se naslednje poddrevo ne začne, dokler se ne »zapre« trenutnega poddrevesa.

Drevo je predstavljeno z bitnim poljem  $B[1 : 2n+2]$ . Zapis temelji na uporabi zapisa stopenj vozlišč, zato je potrebno dodati na začetek bitnega polja  $B[1 : 3] = 110$ . Zatem pa so zapisane stopnje vozlišč z eniškim zapisom. Zapis je zgrajen z uporabo pregleda v globino, kot to ponazarja ime predstavitve. Vsa vozlišča imajo indeks, ki je zaporedno število obiskanega vozlišča. Predstavitev omogoča preprostejše implementacije operacij ter posledično manjše dodatne podatkovne strukture. Vseeno pa omogoča hitrejše implementacije nekaterih operacij v primerjavi z uporabo predstavitve LOUDS. Primer





110|11110|0|110|0|0|110|0|0|0

Slika 4: Primer predstavitve drevesa z metodo DFUDS za priponsko drevo besede »KOKOŠ\$«.

zapisa topologije drevesa z DFUDS je prikazan na Sliki 4. Vozlišča so v zaporedju ločena s sivimi črtami.

Tabela 2: Implementacija operacij drevesa z DFUDS

Operacija	Implementacija v DFUDS
$koren()$	4
$jeList(v)$	$B[v] == 0$
$stOtrok(v)$	$naslednik_0(B, v) - v$
$otrok(v, i)$	$zapri(B, naslednik_0(B, v) - i) + 1$
$prviOtrok(v)$	$naslednik_0(B, v) + 1$
$nBrat(v)$	$iskanjeNaprej(B, v - 1, -1) + 1$
$pBrat(v)$	$zapri(B, odpri(B, v - 1) + 1) + 1$
$starš(v)$	$predhodnik_0(B, izbira_1(B, v - 1)) + 1$
$globina(v)$	/
$lca(v, w)$	$starš(rmq(B, naslednik_0(B, w), v - 1) + 1); v < w$

V Tabeli 2 so predstavljene implementacije operacij nad drevesom, implementirane z DFUDS. Indeks  $izbira_0(B, v) + 1$  je uporabljen za shranjevanje dodatnih informacij o vozlišču, saj je vozlišče  $v$  predstavljeno s položajem vozlišča v bitnem polju  $B$ . Operacija  $zapri(B, i)$ , ki zapre trenutno poddrevo, je definirana z uporabo operacije  $višek$  tako, da vrne prvi  $j > i$ , pri čemer je  $višek(B, j) = višek(B, i) - 1$ . Podobno se lahko definira tudi operacijo  $odpri(B, i)$ , ki vrne koren trenutnega poddrevesa, z uporabo operacije  $višek$  tako, da vrne zadnji  $j < i$ , pri čemer  $višek(B, j) = višek(B, i) + 1$ . V tabeli je vidno, da operacija  $globina(v)$  ni podprta, saj ni mogoče implementirati te operacije brez sprehoda od vozlišča  $v$  do korena ter pri tem šteti potrebne korake.

Operacija  $lca(v, u)$  je lahko implementirana brez sprehoda po drevesu za razliko od implementacije z LOUDS.

Vse operacije, ki temeljijo na operaciji *rang* in *izbira*, se izvršijo v  $O(1)$  času. Operacije, ki temeljijo na *rmM*-drevesu, pa potrebujejo  $O(\log n)$  časa. Pri tem so potrebne dodatne podatkovne strukture za *izbira*<sub>1</sub>, *izbira*<sub>0</sub>, *rang* ter *rmM*-drevo, pri čemer vsaka potrebuje  $o(n)$  dodatnih bitov. Prostorsko zahtevnost *rmM*-drevesa je mogoče zmanjšati, tako da se shranita zgolj polji  $e$  in  $m$ . Tako se razpolovi prostorsko zahtevnost *rmM*-drevesa, ki pa še vedno zahteva  $o(n)$  dodatnih bitov, brez škode za implementacijo operacij drevesa.

Za lažjo implementacijo dodatnih operacij nad listi je mogoče ustvariti dve dodatni podatkovni strukturi za *rang*<sub>00</sub> in *izbira*<sub>00</sub> v konstantnem času, ki omogočata iskanje in indeksiranje listov v drevesu. Podatkovni strukturi sta implementirani na podoben način kot podatkovni strukturi za *rang*<sub>0</sub> in *izbira*<sub>0</sub>, pri tem pa *rang*<sub>00</sub> in *izbira*<sub>00</sub> temeljijo na številu ponovitev oziroma položaju  $i$ -te ponovitve dveh zaporednih bitov z vrednostjo 0.

### 2.5.3 Uravnoreženi oklepaji

Zadnji način zapisa topologije drevesa je zapis z zaporedjem uravnoreženih oklepajev (BP). Drevo se predstavi kot bitno polje  $B$  dolžine  $2n$ , pri čemer je  $n$  število vozlišč v drevesu. Zaporedje se zgradi z uporabo pregleda v globino. Ko je vozlišče prvič obiskano, se na konec do sedaj zapisane sekvence zapiše '('. Uklepaj je v bitnem polju predstavljen s številom 0. Ko se zapiše celotno poddrevo vozlišča, pa se na konec sekvence zapiše ')'. Zaklepaj pa je v bitnem polju zapisan s številom 1. Primer priponskega drevesa, ki je predstavljen z uporabo zaporedja uravnoreženih oklepajev, je prikazan na Sliki 5. Na sliki so vsa notranja vozlišča obarvana z različnimi barvami za lažje prepoznavanje le-teh v zaporedju uravnoreženih oklepajev. Ker so listi oštevilčeni, so ta števila tudi zapisana nad vsakim listom v zaporedju uravnoreženih oklepajev [6].

Torej je vsako vozlišče predstavljeno kot par '(' in ')'. Tako je mogoče definirati sledeče operacije nad oklepaji: *odpri*, *zapri* ter *oklepa*. Operacija *odpri*( $B, i$ ) vrne položaj ')', ki odpre '(' na  $i$ -tem mestu v  $B$ . Operacija *zapri*( $B, i$ ) pa v tej notaciji predstavlja ')', ki zapira '(' na  $i$ -tem mestu  $B$ . Operacija *oklepa*( $B, i$ ) pa vrne položaj  $j$  od '(' v bitnem polju  $B$ , pri čemer velja, da  $j < i < \text{zapri}(B, j)$ . Z uporabo operacije *višek* operacija *OklepaBi* vrne položaj največjega  $j < i$ , pri čemer je  $\text{višek}(B, j - 1) = \text{višek}(B, i) - 1$  [6].

Podobno kot pri predstavitvi drevesa LOUDS, tudi predstavitev BP potrebuje dodatno podatkovno strukturo za operaciji *izbira* in *rang*. Pri tem sta potrebni zgolj podatkovni strukturi za 0, saj operaciji *rang*<sub>1</sub> in *izbira*<sub>1</sub> nista potrebni za pravilno delovanje drevesa v tej predstavitvi. Podatkovni strukturi potrebujeta vsaka  $o(n)$  do-



Slika 5: Primer predstavitve drevesa z metodo BP za priponsko drevo besede »KOKOŠ\$«.

datnih bitov in operaciji se izvršita v konstantnem času.

Tabela 3: Implementacija operacij drevesa z BP

Operacija	Implementacija v BP
$koren()$	1
$jeList(v)$	$B[v] == 0 \wedge B[v + 1] == 1$
$stOtrok(v)$	$minŠtetje(B, v, zapri(B, v) - 2)$
$otrok(v, i)$	$minIzbira(B, v, zapri(B, v) - 2, i) + 1$
$prviOtrok(v)$	$v + 1$
$nBrat(v)$	$zapri(B, v) + 1$
$pBrat(v)$	$odpri(B, v - 1)$
$starš(v)$	$oklepa(B, x)$
$globina(v)$	$2 \cdot rang_0(B, v) - v$
$lca(v, w)$	$oklepa(B, rmq(B, v, w) + 1); v < w$

V Tabeli 3 so prikazane implementacije operacij, ki so potrebne za pravilno delovanje drevesa. Indeks  $izbira_0(B, v)$  je uporabljen za pridobiti dodatne informacije o vozlišču, saj vrednost  $v$  predstavlja položaja '(' v zaporedju  $B$ , ne pa indeksa v tabeli z dodatnimi informacijami o vozlišču.

Zapis topologije drevesa s BP omogoča dodatne operacije. Primer operacije je oštevilčenje in iskanja listov drevesa, kar je storjeno z uporabo posplošitve operacij ranga in izbire na poljubno dolge nize. Ker imajo listi obliko »()« (oziroma 01, ko so zapisani v bitnem polju  $B$ ), se lahko implementirata operaciji  $rang_{01}(B, i)$  in  $izbira_{01}(B, i)$ . Operaciji potrebujeta konstanten čas, da se izvedeta, saj se lahko izgradi podobno dodatno strukturo, kot za osnovno verzijo ranga in izbire. V BP ni mogoče izbrisati

vrednosti  $M$  in  $n$  iz listov  $rmM$ -drevesa, saj se uporablja poizvedbo  $rMq(B, i, j)$ , ki potrebuje vrednost  $M$  pri implementaciji dodatnih operaciji. Primer take operacije je *najglobljeVozlišče*( $v$ ), ki vrne najgloblje vozlišče v poddrevesu s korenem  $v$  [6].

### 3 PRIPONSKA DREVEŠA

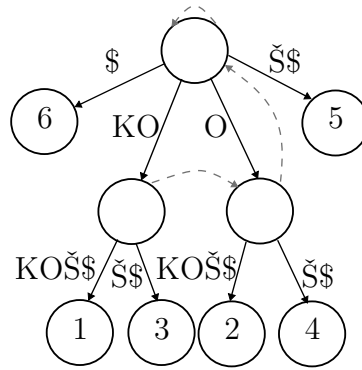
Priponska drevesa so posebna implementacija številskega drevesa, pri čemer vsak list predstavlja posamezno pripono besede. Na ta način priponsko drevo ne predstavlja zgolj vhodne besede  $T$  dolžine  $n$ , ki je sestavljeno iz znakov abecede  $\Sigma$  in je shranjena v pomnilniku kot polje črk  $T[1:n]$ , ampak zakodira tudi strukturo besede. Zato se pogosto uporabljajo za indeksiranje besede in posledično iskanja vzorcev v njej. Vzorec  $P[1 : m]$  se nahaja v besedi  $T$ , če obstaja podniz  $T[i : j]$ , za katerega velja  $P[1 : m] = T[i : j]$ . Z uporabo indeksa je iskanje prisotnosti vzorca  $P$  dolžine  $m$  v besedi primerljivo s KMP algoritmom. KMP algoritem potrebuje  $O(n + m)$  časa za preverjanje, ali se vzorec nahaja v besedi. Za razliko od KMP algoritma je prednost uporabe priponskega drevesa, da lahko iščemo različne vzorce v isti besedi  $T$ . Za iskanje več vzorcev priponskem drevesu je potrebno:  $O(n)$  časa za izgradnjo priponskega drevesa ter  $O(m)$  časa za vsak iskani vzorec. Pri iskanju več vzorcev v besedi  $T$  algoritem KMP potrebuje  $O(n + m)$  časa za vsak iskan vzorec [7, 8].

Nad priponskim drevesom lahko definiramo funkcijo  $beseda(v)$ . Funkcija vrne niz, ki je pridobljen s stikom podnizov na povezavah na poti od korena do vozlišča  $v$ . Torej priponsko drevo nad besedo  $T$ , ki je niz nad abecedo  $\Sigma$ , definiramo na naslednji način [8]:

**Definicija 3.1.** Priponsko drevo nad nizom  $T$  dolžine  $n$  je številsko drevo, ki zadošča naslednjim zahtevam:

1. drevo ima natanko  $n$  listov oštevilčenih s števili med 1 in  $n$ , ki so shranjeni v polju *indeks*,
2. vsako notranje vozlišče, razen korena, ima vsaj dva otroka,
3. vsaka povezava predstavlja neprazni podniz besede  $T$ ,
4. ne obstajata povezavi, ki se začneta v istem vozlišču in z istim znakom,
5. za vsak list  $l$  velja, da je  $beseda(l) = T[l.indeks : ]$ .

Primer priponskega drevesa besede »KOKOŠ« je prikazan na Sliki 6. Znak »\$«, ki predstavlja konec besedila, omogoča bistveno poenostavitev podatkovne strukture, saj tako ni nobena pripona predpona druge pripone. Posledično je vsaka pripona shranjena v listu priponskega drevesa. V primeru, predstavljenem na Sliki 6, znak »\$« ne bi bil



Slika 6: Primer priponskega drevesa nad besedo »KOKOŠ\$«.

potreben, ker že znak »Š« jasno določi vse pripone besede »KOKOŠ«. Da zagotovimo, da so vse pripone shranjene v listih, se besedi na konec pripne znak »\$«. Torej bo v nadaljevanju magistrske naloge znak  $T[n] = \$$ .

Ker povezave v priponskih drevesih predstavljajo podnize v besedi  $T$ , se lahko nad priponskimi drevesi definira tudi črkovna globina vozlič.

**Definicija 3.2.** Črkovna globina vozlišča  $sd(v) = |beseda(v)|$ .

Primer razlike med globino vozlišča in črkovno globino vozlišča se lahko vidi na Sliki 6: vozlišči, do katerih se pride s povezavama »KO« in »O«, imata globino 1. Črkovna globina vozlišča, v katerega kaže povezava »KO«, je 2, medtem ko ima vozlišče, v katerega kaže povezava »O«, črkovno globino 1.

### 3.1 POIZVEDBE

Časovno učinkovito iskanje vzorcev v vhodni besedi  $T$  je doseženo z izgradnjo priponskega drevesa. Zato bodo v tem podpoglavju predstavljene implementacije poizvedb za iskanje vzorcev nad besedo  $T$  z uporabo priponskega drevesa. Obstajajo tri poizvedbe, in sicer:

1.  $prisotnost(T, P)$ , ki preveri, ali je vzorec  $P$  prisoten v besedi  $T$ ,
2.  $številoPonovitev(T, P)$ , ki vrne število ponovitev vzorca  $P$  v besedi  $T$ , in
3.  $seznamPojavov(T, P)$ , ki vrne seznam indeksov v besedi  $T$ , kjer se pojavi vzorec  $P$ .

Osnovna poizvedba nad besedo  $T$  je  $prisotnost(T, P)$ , saj sta ostali dve poizvedbi nadgradnji le-te. Osnovna ideja iskanja vzorcev s priponskimi drevesi je obstoj vzorca  $P$  na začetku vsaj ene pripone besede  $T$  natanko tedaj, ko je  $P$  prisoten v  $T$ . Oziroma,

vzorec  $P[1 : m]$  je prisoten v besedi  $T$  natanko tedaj, ko obstaja pripona  $T[i : n]$ , za katero velja  $P = T[i : i + m]$  ( $P$  je predpona pripone  $T[i : n]$ ).

Listi priponskega drevesa predstavljajo pripone besede  $T$ , zato je poizvedba *prisotnost*( $T, P$ ) z uporabo priponskega drevesa implementirana s sprehodom iz *korena* drevesa proti listom. Ker vsaka povezava predstavlja podniz besede  $T[k : p]$ , ki je lahko shranjen kot par indeksov  $k$  in  $p$ , je potrebno preveriti, ali se  $P$  ujema s  $T[k : p]$ . Če se vzorec ne ujema s podnizom, potem vzorec ni prisoten v besedi in zato poizvedba vrne *false*. Ko pa se  $P$  ujema s podnizom, potem je vzorec prisoten v besedi in zato poizvedba vrne *true*. Tako predstavljen način iskanja deluje zgolj za vzorce, ki niso daljši od  $p - k$  znakov. Če pa je vzorec  $P$  daljši, se najprej pogleda prvih  $p - k$  znakov. Če se podniza ujemata, se nadaljuje z iskanjem na naslednji povezavi, ki se začne v vozlišču  $v$ , do katerega smo prišli po povezavi  $T[k : p]$  na poti proti listom. Pri tem si je potrebno zapomniti, koliko znakov smo že pregledali, in to označimo z  $o$ . V vsakem koraku povečamo  $o$  za  $p - k$  ter preverimo, ali je  $P[o + 1 : o + p - k] = T[k, p]$ . Poizvedba se konča na povezavi, za katero velja, da je  $p - k \geq m - o$ , če se je vseh  $o$  do takrat pregledanih znakov ujemalo. V vozlišču  $v$  je izbrana povezava, za katero velja, da se prvi znak povezave ujema z znakom  $P[o + 1]$ . Iskanje te povezave vzame od  $O(1)$  časa (vsako vozlišče ima polje velikosti  $|\Sigma|$ , tako da ima vsak znak dodeljen prostor za svojo povezavo, čeprav otrok ne obstaja) do  $O(|\Sigma|)$  časa (vsako vozlišče ima povezan seznam povezav do otrok), kar je še vedno konstanten čas, saj se abeceda  $\Sigma$  ne spreminja skozi postopek izgradnje in poizvedb.

Operacija *prisotnost*( $T, P$ ) potrebuje  $O(m)$  časa, da preveri prisotnost vzorca v besedi. Operacija mora preveriti, ali se vsi znaki vzorca ujemajo z znaki na povezavah, ki so del poti od *korena* proti listom, za kar potrebuje  $O(m)$  časa. V vsakem notranjem vozlišču na poti pa potrebuje še dodatno  $O(1)$  časa, da najde naslednjo povezavo. Ker je notranjih vozlišč na poti največ  $m$ , potem tudi celotna poizvedba potrebuje  $O(m)$  časa. Iz tega sledi izrek.

**Izrek 3.3.** *Poizvedba  $prisotnost(T, P)$ , ki je implementirana s priponskim drevesom, potrebuje  $O(m)$  časa in  $O(n)$  prostora.*

Za primer iskanja vzamemo priponsko drevo na Sliki 6, ki predstavlja besedo  $T = \text{«KOKOŠ»}$ . V besedi želimo preveriti prisotnost vzorca  $P_1 = \text{«KOŠ»}$ , ki je prisoten v besedi, ter vzorca  $P_2 = \text{«KOT»}$ , ki pa ni prisoten v besedi. Pri tem predpostavimo, da ima vsako vozlišče fiksno polje kazalcev na otroke (vsaka črka abecede ima eno celico v polju). Iskanje se začne v vozlišču *koren*. Preveri se, ali velja  $\text{koren.otroci['K']} \neq \text{NIL}$ . Ker  $\text{koren.otroci['K']}$  kaže na notranje vozlišče, ki ga imenujemo  $v$ , se lahko preveri, ali se podniz na povezavi ujema s  $P_1$  oziroma  $P_2$ . Ker se oba vzorca začneta s «KO» in se ujemata s podnizom na povezavi, si zapomnimo, da smo pregledali dve črki, in nadaljujemo s pregledovanjem. Za vzorec  $P_1$  preverimo

v vozlišču  $v$ , ali velja  $v.otroci['\check{S}'] \neq NIL$ . Ker obstaja taka povezava, lahko nadaljujemo z iskanjem, ampak smo že preverili vse črke vzorca, ker sta bila do vozlišča  $v$  že pregledana  $o = 2$  znaka in za iskanje naslednje povezave se je pregledal tudi znak  $P_1[3] = \check{S}$ . Zato lahko trdimo, da je vzorec  $P_1$  prisoten v besedi »KOKOŠ«. Za razliko od vzorca  $P_1$  se za vzorec  $P_2$  v vozlišču  $v$  preveri, ali velja  $v.otroci['T'] \neq NIL$ . Ker ne obstaja povezava, ki predstavlja podniz s prvim znakom 'T', vzorec  $P_2$  ni prisoten v besedi »KOKOŠ«.

Preostali poizvedbi sta si zelo podobni in imata isto osnovno idejo implementacije. Brez škode za splošnost lahko uporabimo poizvedbo  $\text{števil}Ponovitev(T, P)$ , da razložimo idejo. Vzorec se nahaja na začetku sprehoda od *korena* proti listom, zato je število ponovitev vzorca v besedi enako številu listov v drevesu, katerih pot do njih se začne z vzorcem  $P$ . Isto velja tudi za poizvedbo  $\text{seznamPojavov}(T, P)$ , pri čemer pa indeksi pripon, ki so shranjeni v listih, predstavljajo indekse v besedi, kjer se pojavi vzorec  $P$ .

Začetka implementacije obeh poizvedb sta enaki kot pri poizvedbi  $\text{prisotnost}(T, P)$ . Če je vzorec  $P$  prisoten v besedi, potem se vzorec  $P$  konča na povezavi, ki vodi v vozlišče  $v$ . Zato velja, da so vsi listi priponskega drevesa, ki predstavljajo pripone s predpono  $P$ , tudi listi v poddrevesu s korenem v vozlišču  $v$ . Torej za najti oziroma prešteti vse ponovitve vzorca  $P$  v vhodni besedi  $T$  se je potrebno sprehoditi po poddrevesu. Poizvedba  $\text{števil}Ponovitev(T, P)$  vrne število obiskanih listov v poddrevesu s korenem v  $v$ , če pa vzorec  $P$  ni prisoten v besedi, pa vrne 0. Poizvedba  $\text{seznamPojavov}(T, P)$  pa vrne seznam vseh indeksov pripon, ki so predstavljeni z obiskanimi listi v sprehodu, če pa vzorec  $P$  ni prisoten v besedi, pa vrne prazen seznam  $[]$ .

Časovna zahtevnost obeh poizvedb je  $O(m + occ)$ , pri čemer je  $occ$  število pojavov vzorca v besedi  $T$ . Ker je najprej potrebno preveriti, ali je vzorec prisoten v drevesu, je potrebno izvesti iste korake kot za poizvedbo  $\text{prisotnost}(T, P)$ , za kar je potrebno  $O(m)$  časa. Za najti vse liste v poddrevesu pa je potrebno še  $O(occ)$  časa. Ker je v priponskem drevesu z  $n$ -timi listi  $O(n)$  notranjih vozlišč, potem je v poddrevesu priponskega drevesa z  $occ$  listi  $O(occ)$  vozlišč. Ker tako pregled v globino kot tudi pregled v širino potrebujeta v drevesih z  $n$ -timi vozlišči  $O(n)$  časa, potemtako za najti vse liste poddrevesa potrebujemo  $O(occ)$  časa.

Za primer vzemimo besedo  $T = \text{»KOKOŠ«}$ , za katero imamo zgrajeno priponsko drevo, ki je prikazano na Sliki 6. V besedi želimo preveriti, kolikokrat se pojavi vzorec  $P = \text{»KO«}$  oziroma želimo narediti poizvedbo  $\text{števil}Ponovitev(T, P)$ . Poizvedbo začnemo v *korenu* priponskega drevesa, kjer preverimo, ali obstaja povezava na vozlišče, ki se začne z znakom  $P[1] = 'K'$ . Ker obstaja povezava  $\text{koren.otroci['K']}$ , imenujemo to vozlišče  $v$  in preverimo, ali se preostanek znakov na povezavi ujema z vzorcem. Povezava predstavlja podniz  $T[1 : 2] = \text{»KO«}$ , in ker smo že preverili prvi znak, je potrebno preveriti še drugi znak, ki pa se tudi ujema. Vzorec je dolg dva znaka, zato se preverjanje prisotnosti vzorca ustavi ter nadaljujemo s preštevanjem listov v



poddrevesu s korenem v  $v$ . Brez škode za splošnost lahko uporabimo pregled v globino ter začnemo s pregledom pregledovati v levem poddrevesu. Skrajno levi otrok od  $v$  je list, ki predstavlja predpono z indeksom 1, zato povečamo število obiskanih listov iz 0 na 1. Nato pregledamo še drugega (zadnjega) otroka, ki je tudi list in predstavlja predpono z indeksom 3, zato se poveča število obiskanih listov iz 1 na 2. Ker smo pregledali celotno poddrevo, poizvedba vrne število 2, kar pomeni, da se vzorec  $P = \text{»KO«}$  pojavi v besedi  $T$  dvakrat. Če pa bi želeli poiskati indekse v besedi, kjer se vzorec pojavi, oziroma izvesti poizvedbo  $\text{seznamPojavov}(T, P)$ , bi bil postopek enak. Pri tem bi beležili indekse pripon namesto štetja obiskanih listov. Ob obisku prvega lista bi se na konec praznega seznama vstavil indeks 1, za drugi list pa bi se na konec seznama vstavil še indeks 3. Torej bi poizvedba vrnila seznam  $[1, 3]$ .

## 3.2 GRADNJA

V tem podpoglavju bodo predstavljene različne metode gradnje priponskih dreves. Metode bodo predstavljene v zaporedju od najbolj počasne, ki zgradi priponsko drevo v času  $O(n^3)$ , do najhitrejša, ki zgradi drevo v času  $O(n)$ . Obstajata dve metodi gradnje priponskega drevesa s časovno zahtevnostjo  $O(n)$ : McCreightov algoritem [4] in Ukkonenov algoritem [1]. V nadaljevanju bo bolj podrobno opisan Ukkonenov sproti (angl. *on-line*) algoritem, saj v vsakem koraku podaljša vse pripone v drevesu za en znak. McCreightov algoritem pa ni sproti algoritem, saj v  $i$ -tem koraku doda  $i$ -to najdaljšo pripono.

### 3.2.1 Naivna metoda

Naivna metoda je sproti metoda gradnje priponskega drevesa. Ta metoda v vsakem koraku s prehodom po drevesu podaljša vse pripone za en znak ter doda novo pripono. Metoda v  $i$ -tem koraku gradnje v do sedaj zgrajeno drevo, ki je bilo zgrajeno za podniz  $T[1 : i - 1]$ , doda znak  $T[i]$  ter tako zgradi drevo za podniz  $T[1 : i]$ .

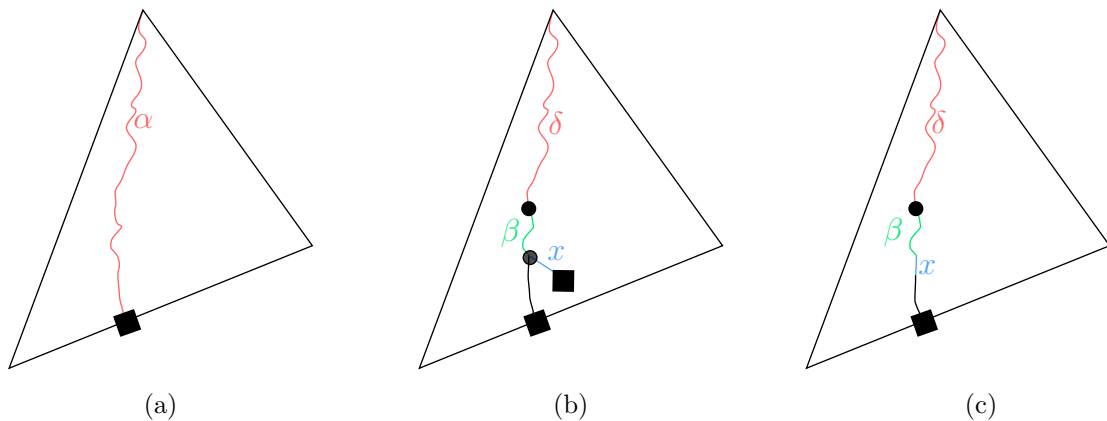
**Načini dodajana** Naj bo  $\alpha$  niz, ki z  $x = T[i]$  tvori niz  $T[k : i] = \alpha x$ , pri čemer je  $1 \leq k \leq i$  in posledično je  $\alpha$  lahko tudi prazen niz. Znak  $x$  je lahko dodan v drevo na tri načine:

1. Če se niz  $\alpha$  konča v listu, potem se zadnja povezava, ki je del niza  $\alpha$ , podaljša za znak  $x$ . Torej enkrat, ko je list zgrajen, ne more postati notranje vozlišče. Način je prikazan na Sliki 7a.
2. Če pa se niz  $\alpha = \delta\beta$  ne konča v listu, potem se konča bodisi v vozlišču  $v$ , bodisi na povezavi med vozliščema, recimo  $v_1$  in  $v_2$ , ter  $\delta$  predstavlja niz iz korena do

vozlišča  $v_1$ ,  $\beta$  pa predstavlja zadnji del niza  $\alpha$  na povezavi med  $v_1$  in  $v_2$ . V tem primeru se lahko znak  $x$  doda na dva načina:

- (a) Če se nobena pot ne nadaljuje z znakom  $x$ , se ustvari nov list  $l$ , na katerega kaže povezava z oznako  $x$ . Če se niz  $\alpha$  konča v vozlišču  $v$ , potem  $l$  postane otrok vozlišča  $v$ . Če pa se  $\alpha$  konča sredi niza na povezavi med vozliščema  $v_1$  in  $v_2$ , se ustvari novo vozlišče  $v'$ . Povezava, ki kaže iz vozlišča  $v_1$  na vozlišče  $v'$ , predstavlja niz  $\beta$ . Iz novega vozlišča kažeta dve povezavi: prva povezava kaže na list  $l$ , druga povezava pa kaže na vozlišče  $v_2$ , ki predstavlja preostanek niza, ki ga je predstavljala predhodna povezava. Način je prikazan na Sliki 7b.
- (b) Če pa obstaja taka pot, ki se nadaljuje po nizu  $\alpha$  z znakom  $x$ , se ne stori ničesar, saj je pripona že prisotna v drevesu kot predpona druge pripone. Način je prikazan na Sliki 7c.

Ti načini podaljševanja pripon veljajo za vse sprotne metode gradnje priponskega drevesa.



Slika 7: Načini dodajanja novih znakov v priponsko drevo. Na sliki kvadratici predstavljajo liste drevesa, krogi pa predstavljajo notranja vozlišča drevesa.

Opazimo, da drevo, ki je zgrajeno za podniz  $T[1 : i]$ ,  $i < n$ , ni nujno priponsko, saj niso vse pripone podniza  $T[1 : i]$  shranjene v listih. Še več, nekatere pripone niso niti eksplicitno predstavljene z vozliščem. Kljub temu pa to drevo vsebuje vso informacijo o pripadajočih priponah, pri čemer moramo posebej beležiti vse pripone, ki so predpone drugim priponam. Takšne pripone imenujemo implicitno predstavljene pripone. Takšnemu drevesu pa rečemo implicitno priponsko drevo. Algoritem gradnje priponskega drevesa z naivno metodo je prikazan v Algoritmu 2. V algoritmu je uporabljena funkcija `najdiVozlišče( $\alpha$ )`, ki vrne najgloblje vozlišče pri iskanju niza  $\alpha$  v drevesu. Primer gradnje priponskega drevesa za besedo »KOKOŠ« z naivno metodo je prikazan



Slika 8: Primer gradnje priponskega drevesa z uporabo Naivne metode za besedo »KOKOŠ\$«.

na Sliki 8. Na Sliki 8 zgolj tretje (zadnje drevo v prvi vrstici) in četrto (prvo drevo v drugi vrstici) drevo sta implicitna priponska drevesa.

**Izrek 3.4.** Naivna metoda zgradi priponsko drevo nad besedo  $T$ , dolžine  $n$ , v času  $O(n^3)$ .

*Dokaz.* Naivna metoda se v vsakem koraku sprehodi čez celotno drevo. Črkovna globina vsakega vozlišča je največ dolžina že dodanega besedila v drevesu, v  $i$ -tem koraku je črkovna globina  $sd(v)$  vsakega vozlišča  $v$  je največ  $i$ . Podobno velja tudi za število listov v drevesu, ki ne presega dolžine že dodanega podniza. Globina lista je manjša ali enaka črkovni globini, torej velja, da se v  $i$ -tem koraku pregleda  $\sum_{j=1}^{i-1} j = \frac{i(i-1)}{2}$  vozlišč.

Ker je niz  $T$  dolg  $n$  znakov, je skozi celotno gradnjo priponskega drevesa število obiskanih vozlišč enako

$$\sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n \frac{i(i-1)}{2} = \frac{n(n+1)(n-1)}{6} = O(n^3).$$

Torej naivna metoda potrebuje  $O(n^3)$  časa. □

**Algoritem 2:** Naivna metoda gradnje priponskega drevesa**Vhod:** Beseda  $T$ , dolžine  $n$ **Izhod:** Priponsko drevo

---

```

1  Ustvari vozlišče koren
2   $s \leftarrow koren$ 
3  za  $i = 1, \dots, n$ 
4      za  $j = 1, \dots, i - 1$ 
5           $v \leftarrow \text{najdiVozlišče}(T[j, i])$ 
6           $u \leftarrow v.\text{otrok}[T[j + sd(v) + 1]]$ 
7          če  $|\text{beseda}(u)| = i - j$  potem
8              sicer če  $jeList(())u$  potem
9                  └─ Uporabi način podaljševanja pripon 1 (strani 23)
10             sicer če  $u.\text{otrok}[T[i]] = NIL$  potem
11                 └─ Ustvari list  $l_i$ ,  $v.\text{otrok}[T[i]] \leftarrow l_i$ 
12             sicer če  $|\text{beseda}(u)[i - j + 1] \neq T[i]$  potem
13                 └─ Uporabi način podaljševanja pripon 2a (strani 23)
14         če  $koren.\text{otrok}[T[i]] = NIL$  potem
15             └─ Ustvari list  $l_i$ ,  $koren.\text{otrok}[T[i]] \leftarrow l_i$ 

```

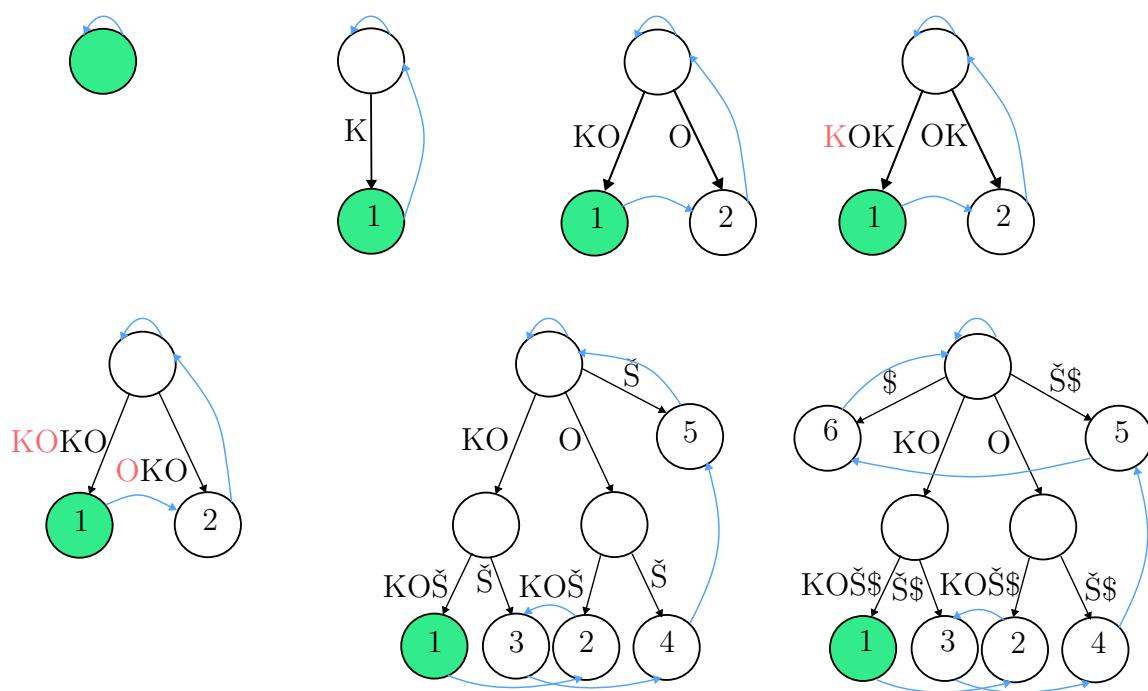
---

**3.2.2 Izboljšana naivna metoda**

Naivna metoda je preprosti način gradnje priponskega drevesa, vendar se hitro opazijo načini za pospešitev. Prvi način izboljšave je pomnjenje implicitnih pripon. Na ta način ni potrebno iskati vsako pripono preden se jo lahko podaljša. Zato lahko naslednji znak dodamo zgolj z enim sprehodom po drevesu. Za tem pa opazimo, da metoda nepotrebno pregleduje celotno drevo. Možna rešitev za ta problem je povezan seznam vozlišč, ki predstavljajo pripone. Vozlišče  $v$ , ki predstavlja pripono, je list, če se pripona konča v listu, sicer pa je vozlišče, iz katerega se začne povezava s koncem pripone.

**Definicija 3.5.** Naj vozlišče  $v$  predstavlja pripono  $T[i : ]$  in vozlišče  $w$  predstavlja pripono  $T[i + 1 : ]$ .  $\Psi$  označimo povezavo, ki kaže iz vozlišča  $v$  na vozlišče  $w$ .

Na Sliki 9, ki prikazuje postopek gradnje priponskega drevesa za besedo »KOKOŠ«, z izboljšano naivno metodo, so prikazane povezave na naslednje vozlišče z modro puščico. Uvedba seznama pripon omogoča izogib nepotrebim sprehodom po drevesu. Tako metoda podaljša vse liste zgolj s sprehodom po seznamu. Pri tem velja, da je  $\Psi(koren()) = koren()$ . Na ta način lahko pregledamo vse pripone v drevesu. Metoda se v vsakem koraku sprehodi iz začetne točke, ki predstavlja najdaljšo pripono,



Slika 9: Primer gradnje priponskega drevesa z uporabo Izboljšane naivne metode za besedo »KOKOŠ\$«.

po seznamu pripon. Na Sliki 9 je začetna točka označena z zeleno barvo. Ker je število vozlišč v seznamu lahko manjše od števila pripon in  $koren()$  kaže nase, metoda šteje, koliko pripon je že podaljšala v trenutnem koraku. Na Sliki 9 so vse implicitno predstavljene pripone, prikazane z rdečo barvo.

Na Algoritmu 3 je prikazana psevdokoda izboljšane naivne metode gradnje. V vsakem koraku gradnje se metoda sprehodi po seznamu vozlišč, ki predstavljajo pripono, in podaljša vse pripone. Vsakič, ko se pripona podaljša z načinom 2a (stran 23), je potrebno popraviti seznam vozlišč. V vsakem koraku je v seznamu največ toliko vozlišč, kot je pripon. Ker je lahko vozlišč manj, metoda šteje število že obiskanih pripon in vse neobiskane pripone se obišče iz korena.

Iz Algoritma 3 je razvidno, da se metoda v  $i$ -tem koraku sprehodi po seznamu vozlišč velikosti  $O(i)$  ter tako podaljša  $i$  pripon. Vhodna beseda je dolga  $n$  znakov, zato je čas gradnje z izboljšano naivno metodo  $O(n^2)$ . Iz tega sledi izrek:

**Izrek 3.6.** *Izboljšana naivna metoda zgradi priponsko drevo nad besedo  $T$  v času  $O(n^2)$ .*

Čeprav te izboljšave znižajo čas gradnje priponskega drevesa iz  $O(n^3)$  na  $O(n^2)$ , le-te ne omogočajo gradnje drevesa v času  $O(n)$ .

**Algoritem 3:** Izboljšana naivna metoda gradnje priponskega drevesa**Vhod:** Beseda  $T$ , dolžine  $n$ **Izhod:** Priponsko drevo

---

```

1  Ustvari vozlišče  $koren$ ;  $ZacetnaTocka \leftarrow koren()$ 
2  za  $i = 1, \dots, n$ 
3       $v \leftarrow ZacetnaTocka$ 
4      za  $j = 1, \dots, i - 1$ 
5          če  $jeList(v)$  potem Uporabi način podaljševanja pripon 1 (strani 23)
6          sicer
7               $k \leftarrow |beseda(v)|$ ,  $u \leftarrow v.otrok[T[j + k + 1]]$ 
8              če  $u = NIL$  potem
9                  Ustvari list  $l_i$ ;  $v.otrok[T[i]] \leftarrow l_i$ 
10                 Popravi seznam pripon ( $v$  spremeni v  $l_i$ )
11             sicer če  $j + |beseda(u)| < i$  potem
12                 če  $u.otrok[T[i]] = NIL$  potem
13                     Ustvari list  $l_i$ ,  $u.otrok[T[i]] \leftarrow l_i$ 
14                     Popravi seznam pripon ( $v$  spremeni v  $l_i$ )
15                 sicer Popravi seznam pripon ( $v$  spremeni v  $u$ )
16             sicer če  $|beseda(u)[i - j + 1] \neq T[i]$  potem
17                 Uporabi način podaljševanja pripon 2a (strani 23)
18                 Popravi seznam pripon ( $v$  spremeni v  $l_i$ )
19          $v \leftarrow v.\Psi$ 
20     če  $koren.otrok[T[i]] = NIL$  potem Ustvari list  $l_i$ ;  $koren.otrok[T[i]] \leftarrow l_i$ 
21     če  $ZacetnaTocka = koren()$  potem  $ZacetnaTocka \leftarrow l_1$ 

```

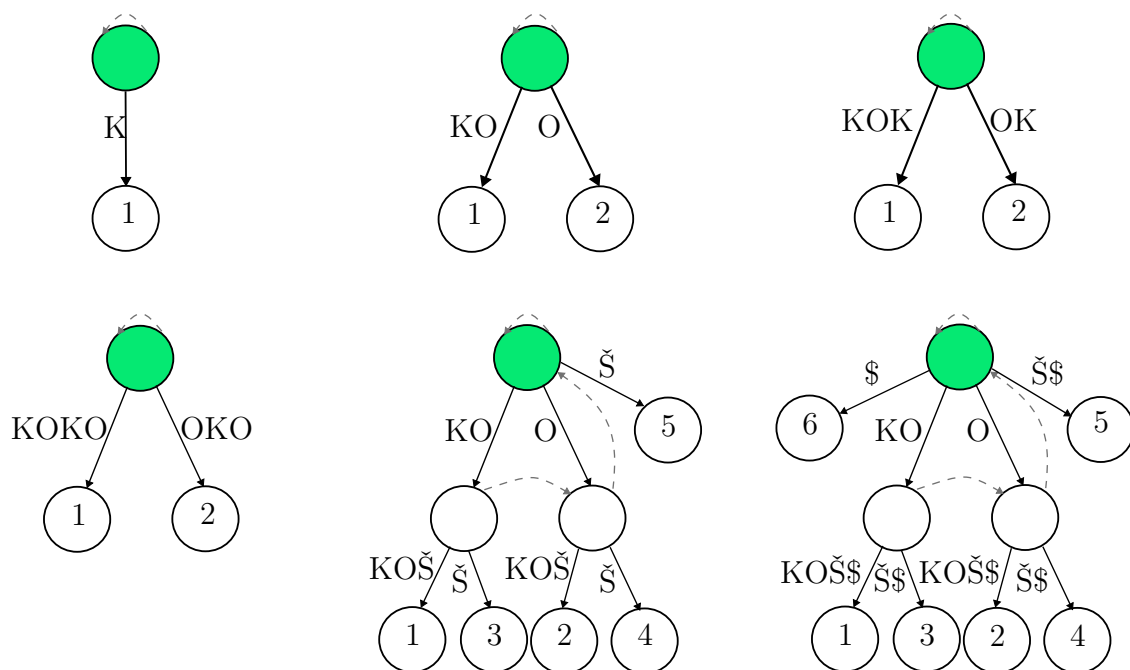
---

**3.2.3 Ukkonenov algoritem**

Ukkonenov algoritem deluje na podoben način kot naivna metoda in izboljšana naivna metoda, saj dodaja v drevo črko po črko. Torej algoritem v  $i$ -tem koraku zgradi priponsko drevo, ki je lahko implicitno priponsko drevo, in predstavlja besedo  $T[1 : i]$ . Algoritem v  $i$ -tem koraku doda v obstoječe drevo črko  $T[i]$ . Črka je dodana v drevo na iste tri načine kot pri ostalih dveh metodah in jih lahko vidimo tudi na primeru gradnje priponskega drevesa za besedo »KOKOŠ\$«. Postopek gradnje z Ukkonenovim algoritmom in vmesna drevesa so prikazana na Sliki 10. Iz načina dodajanja novih znakov v priponsko drevo, ki ga uporablja algoritem, se opazi, da se število vozlišč v drevesu spremeni samo z drugim načinom dodajanja (2a). Torej se moramo sprehoditi zgolj po vozliščih, iz katerih kažejo povezave s konci implicitnih pripon. V teh točkah se

lahko zgodi delitev povezave. Zato se potrebuje dodatno povezavo, ki bo nadomestila povezavo na naslednjo pripono, imenovano priponska povezava (angl. *Suffix link*).

**Definicija 3.7.** Naj bo  $v$  notranje vozlišče, za katerega velja  $beseda(v) = x \cdot \alpha$  ter  $x \in \Sigma$  in  $\alpha \in \Sigma^*$ . Priponska povezava  $sl(v)$  je povezava iz vozlišča  $v$  na notranje vozlišče  $w$ , za katerega velja  $beseda(w) = \alpha$ .



Slika 10: Primer gradnje priponskega drevesa z uporabo Ukkonenovega algoritma za besedo »KOKOŠ\$«.

Podobno kot pri povezavi na naslednjo pripono  $\Psi$ , velja  $sl(koren()) = koren()$ . Na Sliki 10 so priponske povezave označene s sivo črtkano črto. Vse točke, v katerih se bo v tem koraku zgodila delitev, imenujemo aktivne točke (angl. *active point*). Na Sliki 10 je z zeleno označeno vozlišče, ki se nahaja pred začetno aktivno točko. Premik po aktivnih točkah med začetno aktivno točko in končno aktivno točko poteka po priponskih povezavah. Pri tem pa algoritem v vsakem koraku poti izračuna, ali je že prišel v končno aktivno točko. Zato je potrebno hraniti zgolj trenutno aktivno točko ter zastavico, ki hrani vrednost, ali je ta točka tudi končna aktivna točka. Aktivna točka postane končna aktivna točka takrat, ko se način dodajanja spremeni iz 2a v 2b (strani 23), ali, ko so vse implicitne pripone postale listi drevesa, saj od takrat ni več potrebno v trenutnem koraku gradnje deliti povezave in ustvarjati nova vozlišča. Pri tem velja tudi, da končna aktivna točka v koraku  $i - 1$  postane začetna aktivna točka v koraku  $i$ , saj se prva nova delitev v koraku  $i$  lahko zgodi zgolj v točki, kjer so se končale delitve v koraku  $i - 1$ , in to je končna točka koraka  $i - 1$ .

**Algoritem 4:** Ukkonenov algoritem za gradnjo priponskega drevesa**Vhod:** Beseda  $T$ , dolžine  $n$ **Izhod:** Priponsko drevo

---

```

1  Ustvari vozlišče koren
2   $s \leftarrow koren, k \leftarrow 1$ 
3  za  $i = 1, \dots, n$ 
4       $sVoz \leftarrow NIL$ 
5       $(KončnaTočka, v) \leftarrow \text{razdeliTestiraj}((s, (k, i - 1)), T[i])$ 
6      dokler ni KončnaTočka
7          ustvari list  $l$  na katerega kaže točka  $v$ 
8          če  $sVoz \neq NIL$  potem
9              Ustvari priponsko povezavo iz  $sVoz$  v  $v$ 
10              $sVoz \leftarrow v$ 
11              $(s, k) \leftarrow \text{kanoničnaOblika}((sl(s), (k, i - 1)))$ 
12              $(KončnaTočka, v) \leftarrow \text{razdeliTestiraj}((s, (k, i - 1)), T[i])$ 
13     če  $sVoz \neq NIL$  potem
14         Ustvari priponsko povezavo iz  $sVoz$  v  $s$ 
15      $(s, k) \leftarrow \text{kanoničnaOblika}((s, (k, i)))$ 

```

---

Vsako aktivno točko lahko predstavimo kot referenčni par  $(s, \alpha)$ , pri čemer je  $s$  vozlišče pred aktivno točko in  $\alpha$  je niz iz vozlišča  $s$  do aktivne točke. Za lažje shranjevanje niza  $\alpha$  je le-ta shranjen kot par indeksov  $(k, p)$ , pri čemer je  $\alpha = T[k : p]$ . Aktivna točka je lahko predstavljena z različnimi pari  $(s, (k, p))$ . Če je  $s$  najgloblje vozlišče pred aktivno točko, takšen par imenujemo kanonična oblika referenčnega para. Kanonična oblika je po definiciji enolična. Med vsemi referenčnimi pari za dano aktivno točko velja, da je pri paru, ki je v kanonični obliki, niz  $\alpha = T[k : p]$  najkrajši. Na Sliki 10 v četrtem koraku gradnje, ki ga predstavlja prvo drevo v spodnji vrstici, je začetna aktivna točka predstavljena kot  $(koren, (3, 3))$ , v naslednjem koraku pa je začetna aktivna točka  $(koren, (3, 4))$ , ki se bo razcepila in bo postala novo vozlišče. Ta ista točka je lahko po koncu gradnje še vedno predstavljena z referenčnim parom  $(koren, (2, 4))$ , ki pa ni v kanonični obliki. Kanonična oblika te točke je  $(a, (4, 4))$ , pri čemer je vozlišče  $a$  otrok  $koren$ -a, na katerega kaže povezava z nizom »KO«.

Ukkonenov algoritem za gradnjo priponskega drevesa zgradi priponsko drevo s psevdokodo, ki je prikazana na Algoritmu 4. V algoritmu par  $(s, (k, i - 1))$  predstavlja kanonično obliko aktivne točke v trenutnem koraku. Zastavica *KončnaTočka* ima vrednost *true*, če je trenutna aktivna točka tudi končna točka, sicer ima vrednost *false*. Vozlišče  $v$  predstavlja vozlišče, v katerega bo pripet nov list  $v'$ . Povezava do lista  $v'$



bo predstavljala črko  $T[i]$ . Vozlišče  $sVoz$  pa predstavlja vozlišče, na katerega je bil nazadnje pripet list v  $i$ -tem koraku. Vozlišče  $sVoz$  je  $NIL$  zgolj v prvem dodajanju novega lista v vsakem koraku.

Algoritem 4 uporabi dve pomožni funkciji. Prva uporabljena pomožna funkcija je **kanoničnaOblika**, ki za trenutno aktivno točko, predstavljeno z referenčnim parom  $(s, (k, p))$ , vrne njegovo kanonično obliko. Funkcija se sprehodi po drevesu dokler ne doseže najnižjega vozlišča pred aktivno točko. S tem korakom je omogočena učinkovitejša uporaba funkcije **razdeliTestiraj**.

Funkcija **razdeliTestiraj** prejme kot vhod trenutno aktivno točko, ki je podana kot referenčni par  $(s, (k, p))$  v kanonični obliki, ter znak  $t$ , ki se želi vstaviti v priponsko drevo. Funkcija preveri, ali se niza  $T[k : p + 1]$  in  $T[k : p] \cdot t$  ujemata. Če se, potem je trenutna aktivna točka tudi končna točka, zato funkcija ne stori ničesar in vrne  $(true, s)$ . Sicer pa funkcija razdeli povezavo in aktivna točka postane novo vozlišče  $v$ , če že ne obstaja vozlišče  $v$ , nato pa vrne  $(false, v)$ .

**Izrek 3.8.** *Ukkonenov algoritem zgradi priponsko drevo nad besedo  $T$  dolžine  $n$  v času  $O(n)$ .*

*Dokaz.* Dokaz je razdeljen na dva dela: v prvem delu bomo dokazali, da se zanka, ki se začne v vrstici 6, skozi celotno izvajanje algoritma izvede  $O(n)$ -krat, drugi del pa se bo osredotočil na časovno zahtevnost funkcije **kanoničnaOblika**.

Časovna zahtevnost funkcije **razdeliTestiraj** je  $O(1)$ , saj je aktivna točka podana v kanonični obliki, zato ni potrebnih odvečnih sprehodov po drevesu, ki bi povečali časovno zahtevnost. Funkcija preveri, ali je trenutna aktivna točka tudi končna točka, za kar potrebuje konstanten čas. Če ni končna točka, jo spremeni v notranje vozlišče, za kar je potreben konstanten čas. Sicer pa ne stori ničesar in vrne, da je aktivna točka tudi končna točka.

Zanka v  $i$ -tem koraku gradnje dodaja nove povezave na poti iz končne točke  $kt_{i-1}$  koraka  $i - 1$  do končne točke  $kt_i$  koraka  $i$ , ki ni še obiskana. Natančno število obiskanih vozlišč na poti je  $D(kt_{i-1}) - D(kt_i) + 2$ , iz česar sledi, da se s pomočjo seštevne amortizacije v  $n$ -tih korakih zanka izvede

$$\sum_{i=1}^n (D(kt_{i-1}) - D(kt_i) + 2) = D(kt_0) - D(kt_n) + 2n = O(n).$$

Pri tem je potrebno še dokazati, da tudi sprehod v funkciji **kanoničnaOblika** običajno  $O(n)$  vozlišč skozi celotno gradnjo priponskega drevesa. Funkcija ob vsakem klicu pogleda največ  $p - k$  vozlišč, kar je dolžina niza  $\beta = T[k : p]$ . Pri tem pa se niz  $\beta$  z vsakim obiskanim vozliščem skrajša, saj se poveča število  $k$ . Niz  $\beta$  pa se lahko poveča zgolj v 15 vrstici Algoritma 4. Ker se niz  $\beta$  poveča  $n$ -krat skozi celotno gradnjo, se po temtaku tudi niz  $\beta$  lahko zmanjša največ  $n$ -krat skozi celotno gradnjo. Torej funkcija

`kanoničnaOblika` obišče največ  $n$  vozlišč v celotni gradnji priponskega drevesa.

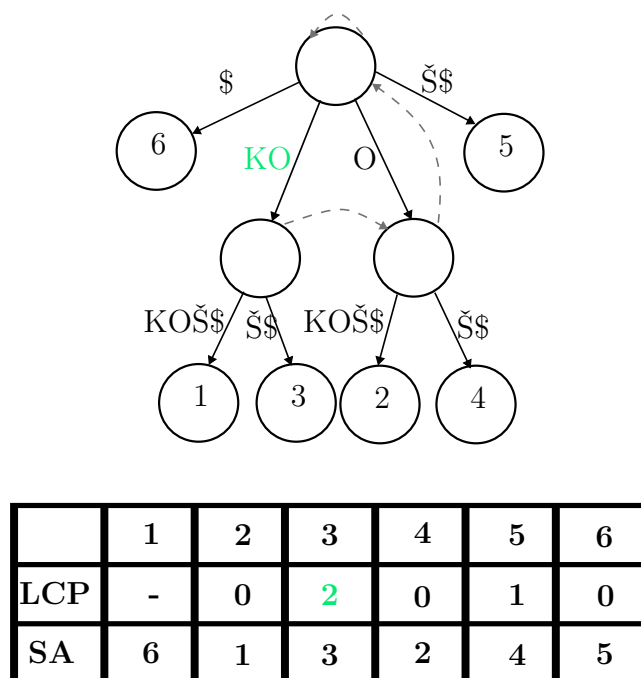
Zanka v vrstici 3 Algoritma 4 se izvede  $n$ -krat, medtem ko se zanka v vrstici 6 in funkcija `kanoničnaOblika` vsaka izvede v  $O(n)$  časa skozi celotno gradnjo priponskega drevesa, iz česar sledi, da tudi gradnja priponskega drevesa potrebuje  $O(n)$  časa, da se izvrši.

□

V nadaljevanju bo uporabljen Ukkonenov algoritem za gradnjo priponskih dreves, ki bodo uporabljena pri empirični analizi v Poglavlju 6, kjer bo tudi izmerjen vpliv pomnjenja notranjega pomnilnika ter posledična uporaba zunanjega pomnilnika (Swap razdelek na zunanjem spominu) namesto notranjega pomnilnika.

## 4 PRIPONSKO POLJE

V priponskem drevesu nad besedo  $T$  dolžine  $n$  na vsako vozlišče, razen na koren, kaže ena povezavo, ki ni priponska. To lahko vidimo na zgornjem delu Slike 11, na katerem je prikazano priponsko drevo za besedo »KOKOŠ\$«, da v vsako vozlišče kaže črna puščica. Vsako vozlišče razen korena ima tudi povezavo na svojega starša. Na Sliki 11 teh povezav ni prikazanih. Vsako notranje vozlišče v priponskem drevesu pa vsebuje tudi priponsko povezavo, le-te pa so na Sliki 11 prikazane s sivo črtkano puščico. Priponsko drevo ima  $n$  listov in  $n_v$  notranjih vozlišč, zato potrebuje  $2(n_v - 1 + n) + n_v = 3n_v - 2 + 2n$  povezav. Priponsko drevo ima največ  $n - 1$  notranje vozlišče, torej priponsko drevo potrebuje med  $2n + 1$  in  $5n - 5$  povezav.



Slika 11: Primer priponskega drevesa in priponskega polja z dodatnim  $LCP$  poljem nad besedo »KOKOŠ\$«.

Vsaka povezava iz vozlišča  $v_1$  v  $v_2$ , pri čemer je  $stars(v_2) = v_1$ , predstavlja neprazen podniz besede  $T$  in sicer podniz  $\alpha$ , ki je definiran kot  $beseda(v_2) = beseda(v_1) \cdot \alpha$ . To se lahko vidi na Sliki 11, saj ima vsaka črna puščica zraven dopisan niz, ki ga predstavlja. Vsak podniz  $\alpha$  je lahko predstavljen kot  $T[s : e]$ . Zato sta v vozlišču  $v_2$  shranjena indeksa  $s$  in  $e$  z dvema celima številoma. Torej potrebuje priponsko drevo še prostor

za shranjevanje  $2(n_v - 1 + n)$  celih števil oziroma med  $2n$  in  $4n - 2$  celih števil.

Vsak list v drevesu predstavlja eno pripono, zato se v vsakem listu hrani še indeks začetka pripone, ki jo predstavlja list, v besedi  $T$ . Z drugimi besedami, če je  $i$  indeks pripone, ki jo predstavlja list  $l$ , potem velja, da je  $beseda(l) = T[i : n]$ . To se lahko vidi tudi na Sliki 11, kjer je v vsakem listu prikazan indeks pripone, ki jo predstavlja. Za predstavitev indeksov začetkov pripon je potrebnih še dodatnih  $n$  celih števil. Če seštejemo vse skupaj, priponsko drevo potrebuje največ  $5n - 5$  referenc na vozlišča in  $5n - 2$  celih števil ali  $10n - 7$  celih števil, če se uporabljajo cela števila kot reference.

Pri tem se pojavi vprašanje, ali je mogoče indeksirati celotno besedo  $T$  zgolj z  $n$  celimi števili. Odgovor je da, saj vsak list v drevesu predstavlja eno pripono. To lahko storimo tako, da shranimo indekse, ki so shranjeni v listih priponskega drevesa, v polju celih števil in sicer polnimo polje iz leve proti desni z indeksi, ki jih dobimo ob prememu sprehodu po drevesu. Pri tem se pojavi problem iskanja po takem polju, saj nam premi sprehod ne zagotavlja leksikografske urejenosti pripon. Brez škode za splošnost lahko predpostavimo, da so povezave, ki kažejo na otroke vozlišč, urejene glede na zaporedje znakov v abecedi  $\Sigma$ . Torej so listi leksikografsko urejeni oziroma je  $beseda(l_i) < beseda(l_{i+1})$ . Ta predpostavka je bila uporabljena na vseh slikah do sedaj. Torej imamo polje indeksov pripon, ki so leksikografsko urejeni, in ga imenujemo priponsko polje (angl. *Suffix array* oziroma SA). Primer priponskega polja za besedo »KOKOŠ« je prikazan na spodnjem delu Slike 11.

V nadaljevanju poglavja bodo predstavljene implementacije poizvedb nad vhodno besedo z uporabo priponskega polja ter pospešitev iskanja z dodatno podatkovno strukturo, imenovano LCP polje. Nato pa bo predstavljen še način gradnje priponskega polja. Za tem bo predstavljena posplošitev LCP polja, ki omogoča simuliranje priponskega drevesa.

## 4.1 POIZVEDBE

Priponsko polje je bilo zasnovano kot alternativa priponskemu drevesu, zato se ga uporablja za indeksiranje besede  $T$  in posledično iskanje vzorcev v njej. V tem podpoglavju bodo predstavljene implementacije istih poizvedb, ki so bile predstavljene za priponsko drevo, vendar s priponskim poljem.

Najbolj osnovna poizvedba, ki bo uporabljena kot osnova za drugi dve poizvedbi, je *prisotnost*( $T, P$ ). Prisotnost vzorca  $P$  dolžine  $m$  v besedi  $T$  lahko preverimo z razpolavljanjem, saj so pripone v priponskem polju urejene. V vsakem koraku razpolavljanja preverimo, ali se pripona na sredini intervala  $[L : R]$  (v prvem koraku je  $L = 1$  in  $R = n$ ) ujema z vzorcem  $P$ , pri čemer označimo indeks te pripone kot  $M$ . Če je  $P = T[SA[M] : SA[M] + m - 1]$ , potem je vzorec  $P$  prisoten v besedi  $T$ , sicer pa

obstaja tak  $k$ , za katerega velja  $P[k] \neq T[SA[M] + k - 1]$ . V tem primeru obstajata dve možnosti:  $P[k] < T[SA[M] + k - 1]$  in zato nadaljujemo z iskanjem v intervalu  $[L : M]$  ali pa  $P[k] > T[SA[M] + k - 1]$  in se nadaljuje z iskanjem v intervalu  $[M : R]$ . Postopek se nadaljuje dokler je  $R - L > 1$  oziroma  $P = T[SA[M] : SA[M] + m - 1]$ . Ko je  $R - L = 1$  in  $P[k] \neq T[SA[M] + k]$  potem  $P$  ni prisoten v besedi  $T$ , sicer pa je vzorec prisoten v  $T$ , saj je  $P = T[SA[M] : SA[M] + m - 1]$ . Tako opisan postopek potrebuje  $O(m \log n)$  časa, saj razpolavljanje potrebuje  $O(\log n)$  primerjav, vsaka primerjava pa potrebuje dodatnih  $O(m)$  primerjav, ali se pripona in vzorec ujemata. Ta način iskanja je  $O(\log n)$ -krat počasnejši od iskanja v priponskem drevesu. To razliko si želimo znižati, pri tem pa ne želimo shraniti celotne topologije drevesa, ampak zgolj informacije, ki pospešijo iskanje po priponskem polju. Informacijo, ki jo želimo shraniti, je število znakov na začetku pripon, ki se ujemajo med dvema priponama, ali z drugimi besedami dolžino najdaljše skupne predpone. To informacijo shranimo v polje najdaljših skupnih predpon (angl. *Longest Common Prefix* oziroma LCP), ki za vsak par indeksov  $i, j$  hrani

$$\begin{aligned} LCP[i, j] &= lcp(T[SA[i], n], T[SA[j], n]) = \\ &= \max\{k \in [1, n]; T[SA[i], SA[i] + k] = T[SA[j], SA[j] + k]\}. \end{aligned}$$

Tako definirano LCP polje potrebuje  $O(n^2)$  prostora. Problem, ki ga želimo rešiti z LCP poljem, je ponovno štetje znakov, za katere vemo, da se ujemajo med pripono  $T[SA[M] : ]$  in  $P$ . Teh je v vsakem koraku  $n_s$ . S tem vedenjem lahko zmanjšamo število primerjav črk med  $P$  in priponami  $T[SA[M] : ]$  na  $O(m)$  skozi celotno izvajanje razpolavljanja.



Slika 12: Prikaz razmerja med  $P$  in priponama  $T[SA[L] : ]$  in  $T[SA[M] : ]$

Na začetku vsakega koraka vemo, da se  $P$  ujema z vsaj  $n_s$  znaki in da smo v levem oziroma desnem podintervalu predhodnega intervala. Recimo, da smo v desnem, torej

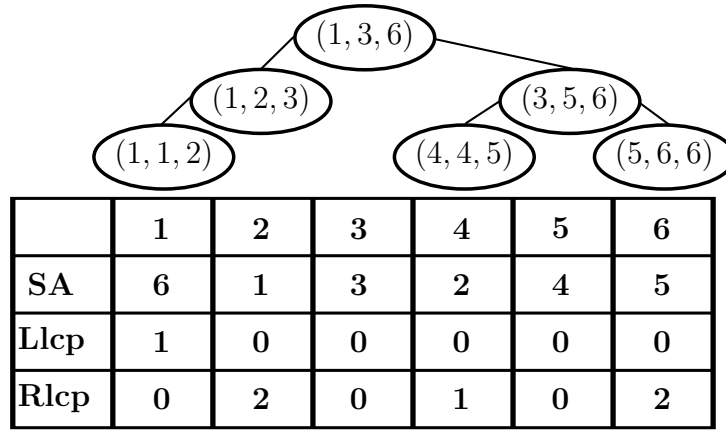
$L$  je bila predhodna sredinska točka. Vemo, da se  $P$  in  $SA[L]$ -ta pripona ujemata v  $k$  znakih ter  $SA[L]$ -ta pripona in  $SA[M]$ -ta pripona se ujemata v  $LCP[L, M]$ -tih znakih ter pripona  $SA[L]$  je leksikografsko manjša od  $SA[M]$ . Torej obstajajo tri možne relacije med  $n_s$  in  $LCP[L, M]$ :

1.  $n_s < LCP[L, M]$ , potem se  $SA[L]$ -ta pripona in  $SA[M]$ -ta pripona bolj ujemata kot  $SA[L]$ -ta pripona in  $P$ . Ker smo v predhodnem koraku izvedeli, da je  $P$  večji od srednje vrednosti predhodnega intervala, to pomeni, da je  $P[n_s + 1] > T[SA[L + n_s + 1]] = T[SA[M] + n_s + 1]$ . Torej je naslednji pregledani interval  $SA[M : R]$ .
2.  $n_s > LCP[L, M]$ , potem se  $SA[L]$ -ta pripona in  $SA[M]$ -ta pripona manj ujemata kot  $SA[L]$ -ta pripona in  $P$ . Označimo  $LCP[L, M] = l$ . Torej  $P[l + 1] = T[SA[L + l + 1]] < T[SA[M + l + 1]]$ , saj je  $L < M$ , ker je  $SA[L]$ -ta pripona leksikografsko manjša od  $SA[M]$ -te pripone. Torej se razpolavljanje nadaljuje na intervalu  $SA[L : M]$ .
3.  $n_s = LCP[L, M]$ , potem je potrebno preveriti, ali je  $P$  večji od  $SA[M]$ -te pripone. Pri tem ni potrebno preveriti prvih  $n_s$  znakov, saj vemo, da se ujemajo, ker se  $SA[L]$ -ta pripona ujema s  $SA[M]$ -ta pripono v  $n_s$  znakih in se  $SA[L]$ -ta pripona tudi ujema s  $P$  v  $n_s$  znakih. Med preverjanjem štejemo, koliko znakov se ujema in to zabeležimo kot  $n'_s$ . Če je  $P[n_s + n'_s + 1] < T[SA[M] + n_s + n'_s + 1]$  potem nadaljujemo v interval  $SA[L : M]$ , sicer je  $P[n_s + n'_s + 1] > T[SA[M] + k + k' + 1]$  in nadaljujemo v interval  $SA[M : R]$ . Preden nadaljujemo v naslednji interval popravimo  $n_s \leftarrow n_s + n'_s$ , pri čemer novi  $n_s \leq m$ . Če je  $n_s = m$  potem je vzorec  $P$  prisoten v besedi  $T$ .

Simetrično velja tudi, če smo v levem podintervalu, pri tem pa uporabimo celico  $LCP[R, M]$ , saj je  $R$  predhodna srednja točka intervala. Iz tega sledi naslednja lema.

**Lema 4.1.** *Poizvedba prisotnost( $T, P$ ), implementirana s priponskim poljem in do sedaj predstavljenim  $LCP$  poljem, potrebuje  $O(m + \log n)$  časa in  $O(n^2 + n)$  prostora.*

Pri tem se opazi, da velika večina celic  $LCP$  polja ne bo nikoli uporabljena pri razpolavljanju. V vsakem koraku razpolavljanja se preverja, ali je vzorec večji od sredinske točke  $M$  na intervalu  $L : R$  indeksov v priponskem polju. Torej za vsak možen interval razpolavljanja je dovolj, da se hrani dolžina najdaljše predpone med  $M$  in  $L$  ter med  $M$  in  $R$ . Ker je vsaka pripona srednja točka natanko enega intervala v razpolavljanju, potem potrebujemo dve  $LCP$  polji, in sicer prvega za shraniti dolžino najdaljše predpone med  $M$  in  $L$ , ki ga imenujemo  $L$ - $LCP$ , in drugega za shraniti dolžino najdaljše predpone med  $M$  in  $R$ , ki ga imenujemo  $R$ - $LCP$ . Primer teh dveh polj je prikazan na Sliki 13, na kateri je prikazano tudi drevo sledi razpolavljanja. Na vsakem



Slika 13: Primer  $L$ -LCP in  $R$ -LCP polji za priponsko polje nad besedo »KOKOŠ\$«.

vozlišču drevesa je prikazan tudi interval indeksov priponskega polja  $(L, M, R)$ , pri čemer  $L$  predstavlja začetek intervala,  $R$  predstavlja konec intervala indeksov in  $M$  predstavlja sredinski indeks, katerega predstavljen pripona bo primerjana z vzorcem  $P$  [24].

Poizvedbo spremenimo tako, da se polje  $LCP$  zamenja z  $L$ -LCP poljem, ko se primerja priponi  $SA[M]$  in  $SA[L]$ , oziroma z  $R$ -LCP poljem, ko se primerja priponi  $SA[M]$  in  $SA[R]$ . Iz tega sledi naslednja lema.

**Lema 4.2.** *Poizvedba  $prisotnost(T, P)$ , implementirana s priponskim poljem in  $L$ -LCP in  $R$ -LCP poljema, potrebuje  $O(m + \log n)$  časa in prostora za  $3n$  celih števil.*

Priponsko polje s to izboljšavo  $LCP$  polja potrebuje manj kot tretjino prostora, ki ga potrebuje ekvivalentno priponsko drevo. Pri tem pa poizvedba  $prisotnost(T, P)$  potrebuje zgolj  $O(\log n)$  dodatnega časa. Opazimo še, da se za vsak interval uporablja bodisi  $L$ -LCP polje bodisi  $R$ -LCP polje, nikoli pa obe polji hkrati. Torej vrednosti, ki bodo uporabljene, se lahko zapišejo v  $Q$ -LCP polje, tako da  $Q$ -LCP $[M] = R$ -LCP $[M]$ , če se primerja priponi  $SA[M]$  in  $SA[R]$ , sicer je  $Q$ -LCP $[M] = L$ -LCP $[M]$ , ko se primerja priponi  $SA[M]$  in  $SA[L]$ . Na ta način se lahko v poizvedbi zamenja  $L$ -LCP in  $R$ -LCP polji z  $Q$ -LCP poljem. Iz tega sledi izrek.

**Izrek 4.3.** *Poizvedba  $prisotnost(T, P)$ , implementirana s priponskim poljem in  $Q$ -LCP poljem, potrebuje  $O(m + \log n)$  časa in prostora za  $2n$  celih števil.*

Idejo poizvedbe  $prisotnost(T, P)$  lahko uporabimo za implementacijo poizvedbe  $številoPonovitev(T, P)$ . Poizvedba vrne število  $occ$ , ki je število ponovite vzorca  $P$  v besedi  $T$ . Naivna implementacija bi bila štetje pripon levo in desno od  $M$ -te pripone, ki je prva pripona v razpolavljanju, ki se ujema z  $P$ . Ta postopek potrebuje  $O(m + \log n + occ)$  časa. Poizvedbo se lahko pohitri na  $O(m + \log n)$  z dvema razpolavljanjema. Prvo razpolavljanje je potrebno za iskanje začetka intervala  $L$  vseh pripon, ki se začnejo s

$P$ , drugo razpolavljanje pa je potrebna za iskanje konca intervala  $R$  vseh takih pripon. Število pripon, ki se začnejo s  $P$ , je  $occ = R - L + 1$ . Vsako razpolavljanje potrebuje  $O(m + \log n)$  časa, torej tudi poizvedba  $številoPonovitev(T, P)$  potrebuje  $O(m + \log n)$  časa.

Na podoben način poizvedba  $seznamPojavov(T, P)$  uporabi interval vseh pripon  $SA[L : R]$ , ki se začnejo s  $P$ , za izdelati seznam indeksov ponovitev vzorca  $P$  v besedi  $T$ . Za najti interval  $SA[L : R]$  je potrebno  $O(m + \log n)$  časa. Za izdelavo seznama indeksov pripon pa je potrebnih še  $occ$  dostopov do priponskega polja oziroma  $O(occ)$  časa. Torej poizvedba  $seznamPojavov(T, P)$  potrebuje  $O(m + \log n + occ)$  časa.

## 4.2 GRADNJA

Podobno kot priponsko drevo je mogoče zgraditi priponsko polje za besedo  $T$  dolžine  $n$  z različnimi algoritmi. Časovna zahtevnost teh algoritmov je med  $O(n^2 \log n)$  in  $O(n)$ . Algoritmi bodo predstavljeni od najbolj neučinkovitega do najbolj učinkovitega.

**Izgradnja s priponskim drevesom:** Priponsko polje je ekvivalentno listom priponskega drevesa, zato se lahko za njegovo izgradnjo uporabi priponsko drevo. Pri tem pa ponovno predpostavimo, da so listi v drevesu leksikografsko urejeni. Priponsko polje je zgrajeno v dveh korakih:

1. izgradnja priponskega drevesa z  $O(n)$  algoritmom, recimo z Ukkonenovim algoritmom (korak ni potreben, če je priponsko drevo že zgrajeno),
2. sprehod v globino po drevesu in zapis indeksov pripon iz listov v polje v vrstnem redu obiska.

Med sprehodom je mogoče zgraditi tudi  $LCP$  polje. Vrednosti v  $LCP$  polju so črkovne dolžine vozlišč, ki so najgloblji predhodnik dveh zaporednih listov. V teh vozliščih se obrne smer sprehoda, saj je do takrat sprehod potekal od lista navzgor in se je v vozlišču obrnil ter bo potekal od vozlišča do lista navzdol.

Opisan algoritem zgradi priponsko polje in  $LCP$  polje v  $O(n)$  časa. Vsak korak potrebuje  $O(n)$  časa, saj smo izbrali algoritem za izgradnjo priponskega drevesa s časovno zahtevnostjo  $O(n)$  in sprehod po drevesu z  $O(n)$  vozlišči potrebuje  $O(n)$  časa. Pri tem pa algoritem potrebuje dodatni prostor za največ  $10n - 7$  števil. Razlog za izgradnjo priponskega polja je zmanjšanje količine spomina, ki ga zasede indeks besede. Zato bi potrebovali algoritem, ki zgradi priponsko polje v  $O(n)$  časa in ne zgradi priponskega drevesa.



**Izgradnja z urejanjem pripon:** Priponsko polje je polje indeksov pripon, urejenih v leksikografskem vrstnem redu, zato se lahko uporabi urejanje za njegovo izgradnjo. Najbolj učinkoviti algoritmi za urejanje (*Quick sort* oziroma *Merge sort*) potrebujejo  $O(n \log n)$  primerjav za izgradnjo priponskega polja. Ker so pripone urejene leksikografsko, vsaka primerjava potrebuje  $O(n)$  časa, torej celotna izgradnja potrebuje  $O(n^2 \log n)$  časa. Pri tem se potrebuje še dodatnega  $O(n)$  časa za izgradnjo *LCP* polja, saj je potrebno izračunati vse *lcp* vrednosti zaporednih pripon [14].

Namesto tega lahko uporabimo korensko urejanje (angl. *RadixSort*). Torej v  $i$ -tem koraku korenskega urejanja so pripone urejene v vedrih glede na prvih  $i$  znakov. To dejstvo se lahko uporabi za izgradnjo *LCP* polja, saj se v  $i$ -tem koraku lahko zapiše vrednost  $i$  na začetek vsakega na novo ustvarjenega vedra. Korensko urejanje potrebuje  $O(kn)$  časa, pri čemer je  $k$  dolžina korena, in v najslabšem primeru je  $k = n$ , torej metoda potrebuje  $O(n^2)$  časa.

Namesto da se v vsakem koraku podaljša koren za en znak, se lahko dolžino korena podvoji. Na ta način potrebuje korensko urejanje  $O(\log n)$  korakov in posledično se potrebuje  $O(n \log n)$  časa za izgradnjo priponskega polja. Pri tem pa vrednost v *LCP* polju na začetku vsakega na novo ustvarjenega vedra ni enaka številu že opravljenih korakov, ampak je v intervalu med  $2^{i-1}$  in  $2^i$ , pri čemer je  $i$  število že opravljenih korakov urejanja. Torej je potrebno poiskati natančno vrednost znotraj intervala s primerjavo znakov, pri tem pa ni potrebno primerjati prvih  $2^{i-1}$  znakov. Algoritem potrebuje  $O(n)$  dodatnega prostora, in sicer 3 polja števil in 2 bitni polji dolžine  $n$  [24].

Obstaja pa bolj učinkovit algoritem za izgradnjo priponskega polja, ki sta ga predlagala Ko in Aluru [25] in potrebuje  $O(n)$  časa. Ideja algoritma temelji na deljenju pripon na dva tipa:  $L$  pripone in  $S$  pripone. Pripona  $T[i : ]$  je  $L$  pripona natanko tedaj, ko je  $T[i : ]$  je manjša od  $T[i + 1 : ]$ , sicer je  $T[i : ]$   $S$  pripona, za katero velja, da je  $T[i : ]$  večja  $T[i + 1 : ]$ . Pri tem velja tudi, da so  $L$  pripone manjše od  $S$  pripon, ki se začnejo z istim znakom  $c$ , torej  $L$  pripone nastopijo pred  $S$  priponami znotraj intervala pripon z istim začetnim znakom. Algoritem zgradi priponsko polje v štirih korakih:

1. deljenje pripon na  $S$  in  $L$ ,
2. ureditev pripon glede na prvi znak pripone,
3. uredi  $S$  pripone ter jih premakni na konec vedra pripon z istim začetnim znakom in premakni konec za eno mesto v levo,
4. dodatno urejanje  $L$  pripon s premikom pripone na začetek intervala, če je pripona, ki je za en znak daljša, tudi  $L$  pripona ter se premakne začetek intervala za eno mesto v desno.

Vsak korak je lahko storjen z enim sprehodom po polju oziroma besedi, ki zahteva  $O(n)$  časa. Podobno kot pri algoritmu s korenskim urejanjem je prostorska zahtevnost tega

algoritma tudi  $O(n)$ , saj se potrebujejo tri polja števil dolžine  $n$  in 3 dodatna bitna polja (dva dolžine  $n$  in enega dolžine  $n/2$ ) [25].

### 4.3 SIMULACIJA OPERACIJ PRIPONSKEGA DREVEŠA

V priponskem drevesu je črkovna dolžina najdaljše skupne predpone dveh pripon enaka črkovni dolžini najglobljega skupnega predhodnika (angl. *Lowest Common Ancestor* oziroma LCA) obeh listov, ki predstavljata priponi. Ampak predhodno predstavljena implementacija  $LCP$  polja je namenjena pospešitvi iskanja po priponskem polju in uporablja  $Q-LCP$  polje. Zato potrebujemo bolj splošno  $LCP$  polje, ki bi nadomestilo  $Q-LCP$  polje in bi omogočalo simuliranje priponskega drevesa.

Če namesto  $Q-LCP$  polja, vzamemo vrednost v  $L-LCP$ , in sicer vrednost  $L-LCP[M]$ , ta predstavlja  $lcp(T[SA[M] :], T[SA[L] :])$  dolžino najdaljše skupne predpone, pri čemer je  $L$  začetni indeks intervala razpolavljanja s sredinsko točko v  $M$ . Označimo vrednost funkcije  $lcp$  kot  $k$ . Vemo tudi, da je  $SA[L]$  leksikografsko manjši od  $SA[M]$ , torej imajo vse pripone na intervalu med  $L$  in  $M$  paroma najdaljšo skupno predpono dolžine vsaj  $k$ , saj so vse pripone na tem intervalu leksikografsko večje od  $SA[L]$  in manjše od  $SA[M]$ . To pomeni, da za vsak  $i$ ,  $L < i \leq M$ , velja  $k \leq lcp(T[SA[i-1] :], T[SA[i] :])$ . Posledično obstaja tak  $i$ , za katerega velja  $lcp(T[SA[i-1] :], T[SA[i] :]) = k$ . Podoben sklep se lahko naredi tudi za  $R-LCP[M]$ , pri čemer uporabimo interval v priponskem polju med  $M$  in  $R$ . Zato se lahko naredi bolj splošno  $LCP$  polje  $LCP[2 : n]$ , ki ima vrednosti

$$LCP[i] = lcp(T[SA[i-1] :], T[SA[i] :]).$$

Primer takega  $LCP$  polja je prikazan na Sliki 11. Na sliki je z zeleno barvo označena vrednost  $LCP[3]$ , ki je najdaljša skupna predpona pripon  $SA[2]$  in  $SA[3]$ . Ta je na sliki označena z zeleno tudi na priponskem drevesu [14, 26].

Novo  $LCP$  polje potrebuje zgolj  $n - 1$  celih števil. Pri tem pa potrebuje dodatno podatkovno strukturo za učinkovito iskanje najmanjše vrednosti na intervalu oziroma  $rmq$ , ki je potrebna za nadomestitev  $Q-LCP$  polja. Prva možnost je izgradnja  $rmM$ -drevesa, ki v vsakem vozlišču vsebuje zgolj vrednost  $m$ . Iskanje v drevesu pa potrebuje  $O(\log n)$  časa oziroma  $O(m \log n + \log n)$  časa za poizvedbo, kar pa je prepočasno, saj iskanje z  $Q-LCP$  poljem potrebuje  $O(m + \log n)$  časa za poizvedbo. Zato lahko uporabimo  $rmq$  strukturo, ki sta jo predlagala Fischer in Heun [28]. Predlagana podatkovna struktura potrebuje  $O(n)$  dodatnega prostora in za dani interval vrne najmanjšo vrednost v  $O(1)$  času, torej se lahko poizvedba izvrši v  $O(m + \log n)$ .

Posplošeno  $LCP$  polje se lahko uporabi tudi za simuliranje priponskega drevesa. Vrednost  $LCP[i]$  predstavlja črkovno dolžino vozlišča  $v$ , za katerega velja  $v = lca(l_{i-1}, l_i)$ , pri čemer  $l_i$  predstavlja pripono  $SA[i]$  in  $l_{i-1}$  predstavlja pripono  $SA[i-1]$ . Ka-

sai idr. [14] so uporabili to dejstvo za simulacijo pregleda od spodaj navzgor (angl. *Bottom-Up Traversal*) in od desne proti levi (angl. *Post-Order Traversal*). S takim obhodom drevesa lahko rešimo problem sprehoda po podnizih (angl. *Substring Traversal Problem*), ki oštevilči vse ponavljajoče se podnize. Podani algoritem potrebuje  $O(n)$  časa in potrebuje enako časa kot obhod priponskega drevesa z  $n$ -timi listi.

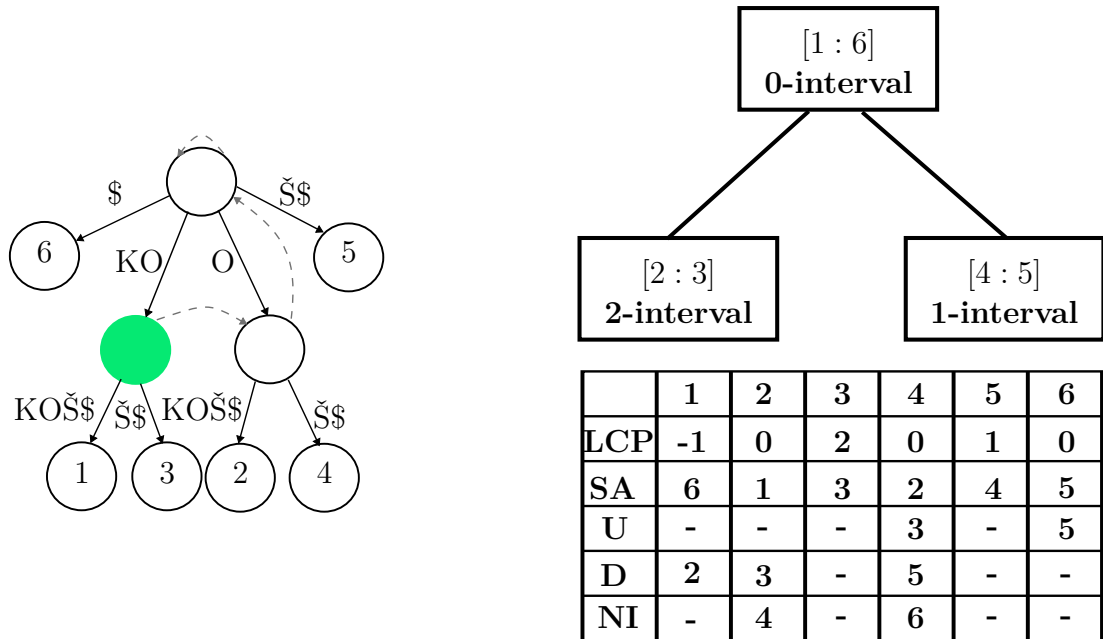
V splošnem vsako notranje vozlišče  $v$  v priponskem drevesu predstavlja podniz besede dolžine  $l$ , s katerim se začnejo vse pripone, predstavljene z listi v poddrevesu s korenem  $v$ . Ker je priponsko polje ekvivalentno listom priponskega drevesa, so vsi listi poddrevesa s korenem  $v$  ekvivalentni intervalu priponskega polja  $SA[L : R]$ , pri čemer je  $SA[L]$ -ta pripona predstavljena s skrajno levim listom,  $SA[R]$ -ta pripona pa s skrajno desnim listom v poddrevesu. Torej so vse vrednosti intervala  $LCP[L + 1 : R]$  vsaj  $l$ . Pri tem velja tudi, da se priponi  $SA[L - 1]$  in  $SA[R + 1]$  zagotovo razlikujeta od pripon  $SA[L]$  in  $SA[R]$  vsaj v  $l$ -tem znaku, sicer bi bile v poddrevesu. Torej velja  $LCP[L] < l$  in  $LCP[R + 1] < l$ . Tak interval  $[L : R]$  je  $l$ -interval, ki je definiran z definicijo 4.4 [26].

**Definicija 4.4.** Interval  $[i : j]$  imenujemo  $l$ -interval v LCP polju, natanko tedaj ko velja:

1.  $LCP[i] < l$ ,
2. obstaja vsaj en  $k$ , za katerega velja, da je  $i < k \leq j$  in  $LCP[k] = l$ ,
3. za vsak  $k$  velja, da je  $i < k \leq j$  in  $LCP[k] \geq l$ ,
4.  $LCP[j + 1] < l$ .

Ker je LCP polje, definirano zgolj za indekse od 2 do  $n$ , in sta v definiciji 4.4 potrebna tudi indeksa 1 in  $n + 1$ , se jih definira kot  $LCP[1] = LCP[n + 1] = -1$ . Ta popravek ne vpliva na pravilnost delovanja priponskega polja, saj ne obstajata priponi  $SA[0]$  in  $SA[n + 1]$ .

Koren priponskega drevesa predstavlja prazen niz in njegovo poddrevo je celotno drevo, zato se koren lahko predstavi kot 0-interval  $[1 : n]$ . Znotraj vsakega  $l$ -intervala  $[L : R]$  lahko obstajajo drugi  $l'$ -intervali, za katere velja  $l < l'$ . Takih  $l'$ -intervalov je največ  $n_l + 1 \leq |\Sigma|$ , pri čemer je  $n_l$  število ponovitev vrednosti  $l$  v LCP polju na intervalu  $[L + 1 : R]$ . Torej lahko predstavimo relacijo med  $l$ -intervali kot LCP intervalno drevo (angl. *LCP interval tree*). Primer LCP intervalnega drevesa je prikazan na Sliki 14. Na sliki se vidi tudi povezavo med notranjimi vozlišči priponskega drevesa in LCP intervalnim drevesom. Na primer, 2-interval  $[2 : 3]$  pokriva priponi, ki se začneta na indeksih 1 in 3. V priponskem drevesu pa sta predstavljeni z listi v poddrevesu zelenega vozlišča [26].



Slika 14: Primer *LCP* intervalnega drevesa nad priponskim poljem in *LCP* poljem besede »KOKOŠ\$« ter njegova predstavitev s tabelo. Na sliki je dodano tudi priponsko drevo besede »KOKOŠ\$«.

**Alternativna predstavitev priponskega drevesa.** Za učinkovito simulacijo operacij nad priponskim drevesom je potrebno učinkovito shraniti *LCP* intervalno drevo. Ker je *LCP* intervalno drevo ekvivalentno notranjim vozliščem priponskega drevesa, se lahko zapiše vsako vozlišče  $v$  kot četvorko  $\langle L, R, l, D \rangle$ , pri čemer interval  $SA[L : R]$  predstavlja vse pripone vozlišča  $v$ ,  $l$  predstavlja dolžino niza vozlišča  $v$  in polje  $D$  (angl. *down* oziroma dol) predstavlja vse otroke vozlišča  $v$ . Če se vsa vozlišča shrani v polje  $V$  dolžine  $n_v + n$ , so v poljih  $D$  shranjeni indeksi v polju  $V$ . Na ta način potrebujemo  $3(n_v + n)$  števil za predstavitev vseh  $R$ -jev,  $L$ -jev in  $l$ -jev ter dodatnih  $(n_v + n - 1)$  indeksov za vozlišča shranjenih v poljih  $D$ . Skupaj potrebujemo  $4(n_v + n) - 1$  števil ter  $n$  števil za priponsko polje, torej potrebujemo največ  $9n - 5$  števil oziroma  $n$  manj števil kot za originalno implementacijo priponskega drevesa.

Polje  $D$  se lahko nadomesti z indeksoma  $D$  in  $NI$  (angl. *next index* oziroma naslednji indeks ali naslednji brat), torej je vsako vozlišče predstavljeno kot peterka  $\langle L, R, l, D, NI \rangle$ , pri čemer  $D$  hrani indeks prvega sina v polju  $V$ ,  $NI$  pa hrani indeks naslednjega brata vozlišča  $v$  v polju  $V$ . Če prvi sin oziroma naslednji brat vozlišča ne obstajata se v  $D$  oziroma  $NI$  shrani vrednost  $-1$ . Tako shranjena vozlišča potrebujejo  $5(n_v + n)$  števil ter  $n$  števil priponskega polja oziroma največ  $11n - 5$  števil.

Opazi se lahko, da je  $L$  od naslednjega brata od  $v$  enak  $R + 1$  od  $v$  oziroma ko je  $NI = -1$  potem je vrednosti  $R$  od starša enaka vrednosti od  $v$ . Torej vrednosti  $R$  ni potrebno hraniti, saj se jo lahko izračuna. Vrednost  $l$  se lahko izračuna s pomočjo

*LCP* polja, kot

$$l = \min_{L < i \leq R} LCP[i],$$

torej se lahko  $l$  nadomesti z *LCP* poljem. Zato si želimo shraniti tudi vrednost  $D$  in  $NI$  v polji dolžine  $n$ . Pri tem se pojavi problem, da je vrednost  $L$  od prvega otroka vozlišča  $v$  enaka vrednosti  $L$  vozlišča  $v$ . Problem se lahko reši tako, da  $D$  kaže na drugega otroka vozlišča  $v$ . Na ta način pa je potrebno še zagotoviti možnost sprehajanja po prvem otroku. To se stori tako, da vozlišče postane četvorka  $\langle L, D, NI, U \rangle$ , pri čemer  $U$  (angl. *up* oziroma *gor*) kaže na drugega sina predhodnega brata. Tako predstavljena vozlišča potrebujejo  $4(n_v + n)$  ter dodatnih  $2n$  števil za priponsko polje in *LCP* polje, oziroma  $10n - 4$  števil.

Na ta način lahko vrednosti v  $D$ ,  $NI$  in  $U$  spremenimo v polja dolžine  $n$  in jih poravnamo z začetki intervala  $L$ . Vrednost  $D[i]$  hrani največji indeks  $j$  večji od  $i$ , za katerega velja  $LCP[j] > LCP[i]$  in vse vrednosti v *LCP* med indeksoma  $i$  in  $j$  so večje od  $LCP[j]$ . Vrednost  $NI[i]$  hrani prvi indeks  $j$  večji od  $i$ , za katerega velja  $LCP[j] = LCP[i]$  in vse vrednosti v *LCP* med indeksoma  $i$  in  $j$  so večje od  $LCP[j]$ . Vrednost  $U[i]$  pa hrani položaj prvega indeksa  $j$  manjšega od  $i$ , za katerega velja  $LCP[j] > LCP[i]$  in vse vrednosti v *LCP* med indeksoma  $j$  in  $i$  so večje ali enake od  $LCP[j]$ . Torej za interval  $SA[i : j]$  velja  $D[i] = U[j + 1]$ . Začetek intervala drugega otroka v prvem podintevalu intervala  $SA[i : j]$  izračunamo kot  $U[D[i]]$ . S temi polji se lahko sprehodimo po priponskem polju, kot bi se sprehodili po priponskem drevesu, pri tem pa potrebujemo  $5n$  števil za shraniti vse te vrednosti [26]. Na Sliki 14 je prikazan tudi zapis intervalnega drevesa s polji  $U$ ,  $D$  in  $NI$ .

**Poizvedbe.** Poizvedba se začne z indeksom  $s = 1$  in  $e = n$  in preverili smo prvih  $o = 0$  znakov vzorca  $P$ . Nato povečamo vrednost  $o$  za ena in najdemo interval, ki se začne z vrednostjo  $P[o]$ , z Algoritmom 5. Nato v vsakem koraku iskanja za trenutni interval  $[s : e]$  poiščemo vrednost  $l = \min_{s < k \leq e} LCP[k]$ . Vemo tudi, da se prvih  $o$  znakov  $P$ -ja ujema s priponami na intervalu  $[s : e]$ . Nato preverimo, ali je  $P[o : k] = T[SA[s] + o : SA[s] + k]$ , pri čemer je  $k = \min\{l, m\}$ . Če se ujemata, popravimo  $o \leftarrow k$  in začnemo iskati podinterval  $[s' : e']$ , za katerega velja  $P[o + 1] = T[SA[s'] + o + 1]$ , ki bo postal novi interval  $[s : e]$ . Če je  $o = m$ , lahko vrnemo, da je  $P$  prisoten v  $T$ , poročamo, da se  $P$  ponovi  $(e - s + 1)$ -krat v  $T$ , ali pa izgradimo seznam indeksov besede  $[SA[s], \dots, SA[e]]$ , odvisno od poizvedbe. Če pa se  $P[o : k]$  ne ujema s  $T[SA[s] + o : SA[s] + k]$ , pa vrnemo, da  $P$  ni prisoten v  $T$ , 0 ali prazen seznam, odvisno od poizvedbe. Iskanje podintervala je implementirano s polji  $D$  in  $U$  za najti prvi podinterval ter s poljem  $NI$  za najti interval, ki se začne z znakom  $P[o + 1]$ . Časovna zahtevnost iskanja podintervala je  $O(|\Sigma|) = O(1)$ , saj se velikost abecede med izvajanjem poizvedbe ne spreminja ter je število podintervalov  $O(|\Sigma|)$ , ker se vsak podinterval razlikuje od drugih vsaj v znaku

na položaju  $l + 1$ . Algoritem za iskanje intervala  $[s' : e']$  prikazuje Algoritem 5 in ne potrebuje  $LCP$  polja, saj smo predhodno že izračunali vrednost  $l$ , ko se je preverjalo prisotnost podniza vzorca  $P[o : k]$ .

---

**Algoritem 5:** Algoritem za iskanje podintervala
 

---

**Vhod:** Začetek intervala  $s$ , konec intervala  $e$ , znak  $c$ , število pregledanih znakov  $l$

**Izhod:** Interval  $[i', j']$

```

1  če  $s < U[e + 1] \leq e$  potem
2     $s' \leftarrow U[e + 1]$ 
3  sicer
4     $s' \leftarrow D[s]$ 
5  če  $T[SA[s] + l + 1] == c$  potem
6     $\text{vrni } [s, s' - 1]$ 
7  dokler  $NI[s'] \neq -1$ 
8     $e' \leftarrow NI[s']$ 
9    če  $T[SA[s'] + l + 1] == c$  potem
10      $\text{vrni } [s', e' - 1]$ 
11    $s' \leftarrow e'$ 
12 če  $T[SA[s'] + l + 1] == c$  potem
13    $\text{vrni } [s', e]$ 
14 vrni  $[-1, -1]$ 

```

---

Torej iskanje s simulacijo operacij nad priponskim drevesom z uporabo priponskega polja in  $LCP$  polja ter  $l$ -intervalov potrebuje  $O(m)$  časa za poizvedbo  $prisotnost(T, P)$  ter omogoča pospešitev časa poizvedbe  $številoPonovitev(T, P)$  na  $O(m)$ . V priponskem drevesu ta poizvedba potrebuje  $O(m + occ)$  časa, saj je potrebno prešteti število listov v poddrevesu. Z uporabo simulacije priponskega drevesa tega štetja ni potrebno storiti, saj poznamo velikost intervala, ki ga pokriva vozlišče, ki je koren poddrevesa, čigar pripone se začnejo z iskanim vzorcem. Število pripon je natanko razlika med začetkom in koncem intervala, pri čemer je konec intervala začetek naslednjega intervala, ki je shranjen v polju  $NI$ . Poizvedba  $seznamPojavov(T, P)$  še vedno potrebuje  $O(m + occ)$  časa, ker je potrebno prehoditi priponsko polje.

**Opomba.** Metoda iskanja z  $LCP$  intervali se lahko uporabi tudi za iskanje po ostalih kompaktnih številskih drevesih (na primer Patricijinih drevesih). Pri tem se nadomesti priponsko polje s poljem listov  $LA$  (angl. *leaf array*), kjer  $i$ -ta celica polja predstavlja besedo, ki jo predstavlja  $i$ -ti list številskega drevesa. Pri tem pa ostaja definicija

$LCP$  polja podobna, in sicer je definirano tako, da so za vsak  $i$  med 2 in  $n$  vrednosti  $LCP[i] = lcp(LA[i - 1], LA[i])$  in vrednost  $LCP[1] = -1$ .

## 5 KOMPAKтна PREDSTAVITEV PRIPONSKEGA DREVEsa

Pri daljših besedah lahko priponska drevesa presežejo velikost notranjega pomnilnika. Prva možna rešitev tega problema je uporaba priponskega polja ter *LCP* polja za simulacijo operacij nad priponskim drevesom. Pri besedah, ki so dvakrat daljše, se problem pojavi tudi pri priponskih poljih. Zato potrebujemo prostorsko bolj učinkovito rešitev.

Vsako vozlišče priponskega drevesa hrani podniz in reference na otroke, starša ter na vozlišče, na katerega kaže priponska povezava. Zato prostor, ki ga zasedejo drevesa v pomnilniku, ni odvisen zgolj od števila vozlišč, temveč tudi od arhitekture računalnika, ki določa velikost pomnilniškega naslova, s katerim se referencira druga vozlišča. Na primer, priponsko drevo človeškega genoma, ki ima dolžino 3 milijarde nukleotidov iz abecede velikosti pet (v abecedo je všteti tudi znak za konec besede, označen z »\$«), potrebuje med 60 in 144 GB notranjega pomnilnika. Ekvivalentno priponsko polje pa potrebuje, odvisno od dodatnih podatkovnih struktur dodanih priponskemu polju, med 12 in 60 GB notranjega pomnilnika. Zato prostorsko učinkovita rešitev ne sme temeljiti na referencah pomnilniških naslovov [13].

Ta problem je mogoče rešiti s kompaktno predstavitvijo podatkovnih struktur. Podatkovna struktura, ki bo predstavljena v tem poglavju, se imenuje kompaktno priponsko drevo (angl. *Compressed Suffix Tree* oziroma CST) in predstavlja eno možno kompaktno predstavitev priponskega drevesa. Preden lahko definiramo kompaktno predstavitev, je treba definirati abstraktno podatkovno strukturo priponsko drevo, ki poda vse operacije, ki so potrebne za pravilno delovanje priponskega drevesa. Večina potrebnih operacij so operacije nad drevesi, ki so bile predstavljene v Definiciji 2.3. Pri tem so dodane tudi specifične operacije, ki so potrebne za pravilno delovanje priponskega drevesa, nekatere operacije pa so popravljene za delovanje z znaki iz abecede  $\Sigma$ . Operacije, ki jih podpira priponsko drevo, so predstavljene v naslednji definiciji [2].

**Definicija 5.1.** Abstraktna podatkovna struktura priponsko drevo nad besedo  $T$  podpira naslednje operacije:



1.  $koren()$ : vrne koren priponskega drevesa,
2.  $jeList(v)$ : vrne *true*, če je vozlišče  $v$  list, sicer vrne *false*,
3.  $otrok(v, z)$ : vrne otroka  $w$  vozlišča  $v$ , katerega povezava se začne z znakom  $z$ .  
Če otrok ne obstaja vrne 0,
4.  $prviOtrok(v)$ : vrne vozlišče  $w$ , ki je prvi otrok vozlišča  $v$ ,
5.  $nBrat(v)$ : vrne vozlišče  $w$ , ki je naslednji brat vozlišča  $v$ ,
6.  $pBrat(v)$ : vrne vozlišče  $w$ , ki je predhodni brat vozlišča  $v$ ,
7.  $starš(v)$ : vrne vozlišče  $w$ , ki je starš od vozlišča  $v$ ,
8.  $povezava(v, i)$ : vrne  $i$ -ti znak na povezavi do vozlišča  $v$ ,
9.  $sd(v)$ : vrne število znakov na poti od korena do vozlišča  $v$ ,
10.  $lca(v, w)$ : vrne najnižjega skupnega prednika vozlišč  $v$  in  $w$ ,
11.  $sl(v)$ : vrne vozlišče  $w$ , na katerega kaže priponska povezava iz vozlišča  $v$ .

V podpoglavju 4.3 je bil predstavljen način simulacije priponskega drevesa z uporabo priponskega polja  $SA$  in polja najdaljših skupnih predpon  $LCP$ . Ideja kompaktne predstavitev priponskega drevesa temelji tudi na teh dveh podatkovnih strukturah. Pri tem pa bomo uporabili kompaktni različici teh podatkovnih struktur. Ker je veliko operacij iz Definicije 5.1 operacij nad drevesi, se lahko doda še kompaktno predstavitev topologije drevesa, saj le-ta pospeši operacije nad drevesi. Iz tega sledi definicija za podatkovno strukturo kompaktno priponsko drevo.

**Definicija 5.2.** Kompaktno priponsko drevo nad besedo  $T$  je sestavljeno iz:

1. kompaktne predstavitev topologije drevesa  $\tau$ ,
2. kompaktne zapisa priponskega polja  $SA$ ,
3. kompaktne predstavitev polja najdaljših skupnih predpon  $LCP$ .

V nadaljevanju poglavja je predstavljena Sadakanejeva [2] implementacija kompaktne priponskega drevesa, ki je prva kompaktna predstavitev priponskega drevesa. V podpoglavju 2.5 je bilo predstavljenih več različnih predstavitev topologije dreves. Implementacija kompaktne priponskega drevesa pa uporablja predstavitev z zaporedjem uravnoveženih oklepajev.

**Topologija drevesa.** Ker se za kompaktno priponsko drevo uporablja BP kot predstavitev topologije drevesa, je potrebno podpirati operacije  $rang_p$ ,  $izbira_p$ ,  $zapri$ ,  $odpri$  in  $oklepa$ , ki se izvajajo v konstantnem času za  $p \in \{0, 1\}^*$ . Pri tem sta potrebni dodatni podatkovni strukturi za operacijo  $rang$  ( $rang_0$ ,  $rang_{01}$ ) in tri za operacijo  $izbira$  ( $izbira_0$ ,  $izbira_1$  in  $izbira_{01}$ ). Operacije  $zapri$ ,  $odpri$  in  $oklepa$  pa uporabljajo podatkovno strukturo za višek (razlika med uklepaji in zaklepaji), ki sta jo predlagala Munro in Raman [15] in je implementirana na podoben način kot dodatna podatkovna struktura za  $rang$ , ki nadomesti  $rmM$ -drevesa. To podatkovno strukturo imenujemo  $L$ , ki hrani polje najnižjih viškov  $L'$  in v celici  $L'[i]$  shrani najnižji višek v  $i$ -tem vedru dolžine  $O(\log n)$ . To pomeni, da je za operacijo  $otrok(v, i)$  potrebno  $O(|\Sigma|) = O(1)$  časa.

Operacija  $lca(v, w)$  pa potrebuje dodatno podatkovno strukturo, ki omogoča konstanten čas izvajanja  $rmq$  poizvedbe nad viškom. Ta podatkovna struktura potrebuje  $o(n)$  dodatnih bitov. Zgradi se dvodimenzionalno tabelo  $M$  in vrednost  $M[i][k]$  hrani položaj najmanjšega viška na intervalu  $L'[i : i + 2^k - 1]$ . Torej operacija  $lca(v, w)$  je izračunana kot

$$\min \{ \min(L[v : e]), M[v'][k], M[w' - 2^k + 1][k], \min(L[s : w]) \},$$

pri čemer je  $k = \lfloor \log(w' - v') \rfloor$ ,  $v'$  označuje vedro v  $L'$ , ki je prvo vedro po vedru z vozliščem  $v$ ,  $w'$  je vedro v  $L'$ , ki je zadnje vedro pred vedrom z vozliščem  $w$ ,  $e$  predstavlja konec vedra v  $L$ , ki vsebuje  $v$ , in  $s$  predstavlja začetek vedra, ki vsebuje  $w$ . Pri tem vsaka poizvedba v  $L$  in vsaka poizvedba v  $M$  potrebuje konstanten čas [2, 3].

V primeru, da so potrebne dodatne operacije, ki temeljijo na  $rmq(\cdot, \cdot)$  operaciji, ali  $|\Sigma| > \log n$ , se lahko doda še  $rmM$ -drevo (angl. *range minimum maximum tree*, predstavljeno v podpoglavju 2.4), ki omogoča izvedbo operacij v  $O(\log n)$  ter pospeši izvajanje oziroma omogoča implementacijo tudi drugih operacij.

**Lema 5.3.** *Podatkovna struktura za predstavitev topologije priponskega drevesa  $\tau$  nad besedo  $T$  dolžine  $n$  potrebuje največ  $4n + o(n)$  bitov.*

*Dokaz.* Priponsko drevo nad besedo  $T$  ima največ  $2n - 1$  vozlišč: od teh je  $n - 1$  notranjih vozlišč in  $n$  listov. Vsako vozlišče se predstavi kot par oklepajev, torej potrebujemo 2 bita za predstavitev vsakega vozlišča. Torej je potrebnih največ  $4n - 2$  bitov za zapis topologije drevesa  $\tau$  kot sekvenco uravnovešenih oklepajev. Zato celotna podatkovna struktura topologije drevesa potrebuje največ  $4n + o(n)$  bitov, saj vsaka dodatna podatkovna struktura potrebuje še dodatnih  $o(n)$  bitov.  $\square$

**Kompaktno priponsko polje.** Naslednja podatkovna struktura, ki je potrebna za pravilno delovanje kompaktne priponskega drevesa, je kompaktno priponsko polje (angl. *Compressed Suffix Array* oziroma CSA). Kompaktna priponska polja znižajo prostorsko zahtevnost priponskega polja iz  $O(n \log n)$  oziroma  $O(nw)$  na  $O(n \log |\Sigma|)$

bitov [16] ali celo na  $nH_h + o(n)$  bitov [17], pri čemer je red  $h \leq \alpha \log_{|\Sigma|} n$ ;  $0 < \alpha < 1$  in  $H_h$  je entropija  $h$ -tega reda, ki se v praksi uporablja kot merilo prostorske zahtevnosti pri kodiranju besedil [6].

Obstajata dve različici kompaktnih priponskih polj: prva implementacija temelji na funkciji  $\Psi$ , druga implementacija, imenovana  $FM$ -indeks pa uporablja  $LF$  funkcijo, ki je inverzna funkcija od  $\Psi$ . Funkcija  $\Psi(i)$  vrne položaj pripone  $T[i + 1 :]$  v priponskem polju, kar pomeni, da

$$\Psi(i) = SA^{-1}[SA[i] + 1],$$

pri čemer  $SA^{-1}[i]$  hrani položaj pripone  $T[i :]$  v priponskem polju  $SA$  in polje  $SA^{-1}[1 : n]$  se imenuje inverzno priponsko polje. Torej se je mogoče s funkcijo  $\Psi$  sprehajamo po priponskem polju oziroma znotraj besede  $T$  iz leve proti desni. Funkcija  $LF$  pa omogoča sprehod v obratni smeri, iz desne proti levi [2, 6].

Implementacije kompaktnega priponskega polja, ki se lahko uporabljajo pri implementaciji kompaktnih priponskih dreves, morajo podpirati sledeče operacije.

**Definicija 5.4.** Kompaktno priponsko polje nad besedo  $T$ , ki se uporablja v kompaktnem priponskem drevesu, podpira sledeče operacije z dano časovno zahtevnostjo:

1. *pripona*( $i$ ): vrne začetni indeks pripone, shranjen v  $SA[i]$ , v času  $t_{SA}$ ,
2. *inverz*( $i$ ): vrne indeks  $j = SA^{-1}[i]$ , pri čemer je  $SA[j] = i$ , v času  $t_{SA}$ ,
3.  $\Psi(i)$ : vrne  $SA^{-1}[SA[i] + 1]$  v času  $t_\Psi$ ,
4. *besedilo*( $i, d$ ): vrne  $T[SA[i], SA[i] + d - 1]$  v času  $O(dt_\Psi)$ .

Funkcija  $\Psi$  je uporabna v kompaktnih priponskih drevesih za gradnjo priponskih povezav v času  $t_\Psi$ , čeprav ni potrebna za pravilno delovanje priponskega polja. Iz definicije funkcije  $\Psi$  je razvidno, da jo je mogoče simulirati z uporabo operacij *inverz*( $\cdot$ ) in *pripona*( $\cdot$ ). Zato je čas, potreben za izvršiti funkcijo  $\Psi$   $t_\Psi \leq t_{SA}$ . Takšen čas namreč potrebuje operaciji *inverz*( $\cdot$ ) in *pripona*( $\cdot$ ). Iz Definicije 5.4 sledi, da mora kompaktno priponsko polje podpirati funkcijo  $\Psi$ , zato implementacije s  $FM$ -indeksom niso primerne, saj morajo funkcijo  $\Psi$  simulirati kot *inverz*(*pripona*( $\cdot$ )).

V nadaljevanju tega dela poglavja je predstavljena preprosta različica kompaktnega priponskega polja. Ta različica je bila izbrana zaradi enostavne implementacije operacij ter preproste vizualizacije podatkovne strukture. V tej implementaciji so vse štiri operacije implementirane z uporabo  $\Psi$  funkcije ter z vzorčenjem priponskega polja  $SA$  in inverznega priponskega polja  $SA^{-1}$ . Na ta način ni treba hraniti besede  $T$  in celotnega priponskega polja  $SA$ .

Funkcija  $\Psi$  je predstavljena z istoimenskim poljem, katerega  $i$ -ta celica je  $\Psi[i] = \Psi(i) = SA^{-1}[SA[i] + 1]$ . Ideja kompaktnega zapisa priponskega polja temelji na dejstvu,

da se lahko premikamo po besedi in posledično med priponami zgolj z uporabo  $\Psi$  funkcije, saj je  $SA[\Psi[i]] = SA[i] + 1$ . Torej se lahko znebimo besede  $T$  in jo zamenjamo z bitnim poljem sprememb  $D$ , pri čemer  $D[i] = 1$ , ko je  $i = 1$  ali  $T[SA[i]] \neq T[SA[i + 1]]$ , ter s poljem znakov  $S$ , ki so urejeni v leksikografskem vrstnem redu, tako da je  $T[SA[i]] = S[rang_1(D, i)]$  [6].

Kompaktni zapis polja  $\Psi$  razdeli polje  $\Psi$  na  $|\Sigma|$  delov in uvede polje  $C$  začetnih točk intervalov v  $\Psi$  polju, ki se začnejo s poljubnim znakom  $c \in \Sigma$ , oziroma  $C[c] = i$  za poljuben znak iz  $\Sigma$ , tako da je  $T[SA[i + 1]] = c$  in  $T[SA[i]] \neq c$ . Polje  $C$  potrebuje  $O(|\Sigma| \log n)$  bitov. Tako se lahko nadomesti polje  $\Psi$  s polji  $\Psi_c$ ;  $c \in \Sigma$ , kjer je  $\Psi[i] = \Psi_c[i']$ , za  $i' = i - C[S[rang_1(D, i)]]$ . Vsako polje  $\Psi_c$  se lahko zapiše kot bitno polje  $B_c$  dolžine  $n$ , pri čemer  $B_c[\Psi_c[i]] = 1$  za  $1 \leq i \leq n_c$ , kjer je  $n_c$  število ponovitev pripon, ki se začnejo z znakom  $c$ , v besedi  $T$ . Torej velja  $\Psi_c[i'] = izbira_1(B_c, i')$  ali  $\Psi[i] = izbira_1(B_c, i - C[c])$ , kjer je  $c = S[rang_1(D, i)]$ . Bitna polja  $B_c$  so zelo redka (število enic je bistveno manjše od števila ničel), torej se jih lahko stisne iz  $n + o(n)$  bitov na  $n_c \log \frac{n}{n_c} + O(n_c)$  bitov, pri čemer ohranimo možnost opravljanja operacije  $izbira_1$  v konstantnem času, kar pomeni, da funkcija  $\Psi$  potrebuje  $nH_0(T) + O(n + |\Sigma|w)$  bitov, pri čemer  $H_0(T)$  je entropija besede  $T$ , in potrebuje  $t_\psi = O(1)$  časa [6].

Do sedaj predstavljena podatkovna struktura omogoča zgolj iskanje števila ponovitev vzorca v besedi, saj omogoča premikanje med intervali s priponami z istim začetnim znakom, pri tem pa ne omogoča določanja lokacije pojavov vzorca v besedi. Za to sta potrebni dodatni podatkovni strukturi, ki nadomestita priponsko polje  $SA$  in inverzno polje  $SA^{-1}$ . Polji je mogoče nadomestiti s poljema vzorcev. Priponsko polje  $SA$  se vzorči s faktorjem  $l = \Theta(\log n)$ -krat, kar je dober kompromis med časovno in prostorsko zahtevnostjo. Pri tem se uporablja dodatno bitno polje  $B[1, n]$ , kjer  $B[i] = 1$  natanko tedaj, ko  $i = 1$  ali  $SA[i] \bmod l = 0$ . Polje vzorcev  $SA_S$  vsebuje vrednosti  $SA_S[rang_1(B, i)] = SA[i]$ , ko je  $B[i] = 1$ . Ostale vrednosti  $SA[i]$  se izračunajo kot  $SA[i] = SA_S[rang_1(B, i_k)] - k$ , pri čemer je  $i_k$  rezultat  $k$ -kratne uporabe funkcije  $\Psi$  nad vrednostjo  $i$  oziroma  $i_k = \Psi^k(i)$  (na primer  $i_2 = \Psi^2(i) = \Psi[\Psi[i]]$ ). Ker je  $k < l$ , je časovna zahtevnost poizvedbe v priponskem polju  $t_{SA} = O(l) = O(\log n)$  [6].

Podobno se vzorči tudi polje inverzov pripon  $SA^{-1}$ , le da je polje  $SA^{-1}$  vzorčeno na enakomernih intervalih dolžine  $l = \Theta(\log n)$ , kar je dober kompromis med časovno in prostorsko zahtevnostjo. Torej ima  $i$ -ta celica polja  $SA_S^{-1}[1, \lfloor n/l \rfloor]$  vrednost  $SA_S^{-1}[i] = SA^{-1}[il]$ . Poljubno vrednost  $SA^{-1}$  se izračuna tako, da se najprej izračuna  $i' = \lfloor i/l \rfloor l$ , nato se  $(i - i')$ -krat uporabi funkcija  $\Psi$  nad vrednostjo  $j' = SA_S^{-1}[i'/l]$  oziroma  $SA^{-1}[i] = \Psi^{i-i'}[j'] = \Psi^{i-i'}[SA_S^{-1}[i'/l]]$ . Isti razmislek kot za časovno zahtevnost dostopa do polja  $SA$  velja tudi za časovno zahtevnost dostopa do polja  $SA^{-1}$ , ki ima časovno zahtevnost dostopa  $t_{SA} = O(l) = O(\log n)$ . Pri tem velja omeniti, da če se inverzno priponsko polje uporablja bolj poredko, se ga lahko vzorči z večjim faktorjem  $l$ , kot se je vzorčilo priponsko polje  $SA$ . Na ta način se zmanjša prostorska zahtevnost,

poslabšajo pa se časovne zahtevnosti operacije, ki se uporabljajo bolj poredko [6].

Tako predstavljeno kompaktno priponsko polje potrebuje  $|\Sigma|n + O(n)$  bitov. Ker pa je večina bitnih polj redkih, saj je število bitov z vrednostjo 0 veliko večje od števila bitov z vrednostjo 1, se lahko bitna polja zakodirajo na bolj kompakten način z enim od algoritmov za stiskanje. Torej stisnjena bitna polja potrebujejo  $nH_0(T) + O(n + |\Sigma|w)$  bitov za implementacijo kompaktne priponskega polja. Potrebni čas za izračun funkcije  $\Psi$  ostaja  $t_\Psi = O(1)$ , čas za iskanje po priponskem polju in inverznem priponskem polju pa je  $t_{SA} = l \cdot t_\Psi = t_\Psi O(\log n) = O(\log n)$ .

**Polje najdaljših skupnih predpon.** Zadnja podatkovna struktura, ki sestavlja kompaktno priponsko drevo, je polje najdaljših skupnih predpon oziroma *LCP* polje. Namesto *LCP* polja se bomo osredotočili na permutacijo *LCP* polja, imenovano *PLCP*, za katero velja relacija  $PLCP[i] = LCP[SA^{-1}[i]]$  ali  $LCP[i] = PLCP[SA[i]]$  in ki jo je lažje predstaviti v kompaktnem zapisu [2, 6].

Ideja kompaktne zapisa *LCP* polja, natančneje *PLCP* polja, temelji na dejstvu, da je  $LCP[i] = LCP[SA^{-1}[i] + 1] - 1$ . Torej, če je to dejstvo mogoče zapisati kot strogo naraščajoče zaporedje, se lahko zaporedje predstavi kot bitno polje  $H$  in vrednost  $H[i] = 1$  za vsako vrednost v zaporedju, sicer je  $H[i] = 0$ . Torej se  $LCP[i]$  lahko izračuna kot  $izbira_1(H, i)$ .

Zaporedje  $PLCP[i] + 2i$  je za vsak  $i$  med 1 in  $n - 1$  strogo naraščajoče, saj je  $PLCP[i + 1] \geq PLCP[i] - 1$ . To lahko enostavno dokažemo, saj iz  $PLCP[i] = 0$  sledi  $PLCP[i + 1] = 0$ . V nasprotnem primeru pa obstajata priponi  $T[j : ]$  in  $T[i : ]$ , kjer je  $T[i : ] < T[j : ]$  in imata najdaljšo skupno predpono dolžine  $PLCP[j] > 0$ . Potem imata  $T[i + 1 : ]$  in  $T[j + 1 : ]$  najdaljšo skupno predpono dolžine  $PLCP[j] - 1$ . Torej je  $PLCP[j + 1] = PLCP[j] - 1$ . Posledično velja, da je  $PLCP[j] + 2j < PLCP[j + 1] + 2(j + 1) = PLCP[j] + 2j + 1$ . Vrednost  $PLCP[n - 1] + 2(n - 1) < 2n$ , saj ima lahko pripona  $T[n - 1 : ]$  dolžino najdaljše skupne predpone s poljubno pripono največ 1. Zato se lahko permutacija *PLCP* in posledično *LPC* polje zapiše kot bitno polje  $H[1 : 2n - 1]$ . Celica  $H[j] = 1$  za  $j = PLCP[i] + 2i$ , kjer je  $i$  med 1 in  $n - 1$ , sicer pa je  $H[j] = 0$ . Pri tem bitno polje  $H$  potrebuje dodatno podatkovno strukturo za  $izbira_1$  [2, 6].

Vrednost  $LCP[i]$  se lahko pridobi v času  $O(t_{SA})$ . Vrednost se izračuna z uporabo formule  $LCP[i] = izbira_1(H, SA[i]) - 2SA[i]$ . Poizvedba potrebuje  $O(t_{SA})$  časa, saj je treba izračunati vrednost  $SA[i]$ , ki se izračuna v  $O(t_{SA})$  časa, ostale operacije pa potrebujejo konstanten čas. Če se uporablja predhodno predstavljena implementacija kompaktne polja, je čas poizvedbe v *LCP* polju enak  $O(\log n)$ . Prostorska zahtevnost polja *LCP* je predstavljena s sledečo lemo:

**Lema 5.5.** *Podatkovna struktura LCP polje potrebuje  $2n + o(n)$  bitov.*

*Dokaz.* Podatkovna struktura  $LCP$  polje se shrani kot bitno polje  $H[1, 2n - 1]$ . Bitno polje  $H$  potrebuje dodatnih  $o(n)$  bitov za podatkovno strukturo, ki omogoča izvajanje operacije  $izbira_1$  v konstantnem času. Ker je bitno polje  $H$  dolžine  $2n - 1$  in potrebuje dodatnih  $o(n)$  bitov, potem celotna podatkovna struktura potrebuje  $2n + o(n)$  bitov za shranjevanje  $LCP$  polja.  $\square$

Sedaj, ko so bile predstavljene vse podatkovne strukture, ki sestavljajo kompaktno priponsko drevo, je mogoče izračunati velikost celotnega kompaktne priponskega drevesa. Ker obstaja več implementacij kompaktne priponskega polja, ki se lahko uporabijo v kompaktnem priponskem drevesu, bo velikost priponskega drevesa vsebovala člen  $|CSA|$ , ki predstavlja velikost kompaktne priponskega polja. Velikost kompaktne priponskega drevesa je predstavljena v sledečem izreku:

**Izrek 5.6.** *Podatkovna struktura kompaktno priponsko drevo nad besedo  $T$  dolžine  $n$  potrebuje  $|CSA| + 6n + o(n)$  bitov, pri čemer  $|CSA|$  predstavlja velikost kompaktne priponskega polja.*

*Dokaz.* Ker obstajajo različne implementacije kompaktne priponskega polja, je potrebnih  $|CSA|$  bitov za shraniti kompaktno priponsko polje  $SA$ . Iz Leme 5.3 sledi, da je potrebnih  $4n + o(n)$  bitov za shraniti topologijo drevesa  $\tau$ . Iz Leme 5.5 pa sledi, da je potrebnih  $2n + o(n)$  bitov za shraniti polje  $LCP$ .

Torej je velikost kompaktne priponskega drevesa  $CST$  enaka  $|CSA| + 6n + o(n)$  bitov.  $\square$

Kot je bilo že omenjeno, obstaja več različnih implementacij kompaktne priponskega polja, zato Sadakane [2] predlaga dve implementaciji. Prva predlagana implementacija je prostorsko učinkovita in uporablja kompaktno priponsko polje, ki so ga predlagali Grossi idr. [17].

**Posledica 5.7.** *Naj bo kompaktno priponsko drevo  $CST$  implementirano z uporabo kompaktne priponskega polja  $CSA$ , ki potrebuje  $|CSA| = nH_h + O(n \log \log n / \log_{|\Sigma|} n)$  bitov. Potem takšno drevo potrebuje  $|CST| = nH_h + 6n + O(n \log \log n / \log_{|\Sigma|} n)$  bitov.*

Druga predlagana implementacija pa je časovno učinkovita in uporablja kompaktno priponsko polje, ki sta ga predlagala Grossi in Vitter [16].

**Posledica 5.8.** *Naj bo kompaktno priponsko drevo  $CST$  implementirano z uporabo kompaktne priponskega polja  $CSA$ , ki potrebuje  $O(\epsilon^{-1} n \log |\Sigma|)$  bitov. Potem takšno drevo potrebuje  $|CST| = O(\epsilon^{-1} n \log |\Sigma|)$  bitov. Pri tem je  $\epsilon$  poljubna konstanta, ki ima vrednost  $0 < \epsilon < 1$ .*

Primer kompaktne priponskega drevesa je prikazan na Sliki 15. Prikazana podatkovna struktura potrebuje približno 12 B (natančneje 97 bitov). Pri tem so cela

	1	2	3	4	5	6
$S$	\$	K	O	Š		
$C$	0	1	3	5		
$D$	1	1	0	1	0	1
$B_1$	0	1	0	0	0	0
$B_2$	0	0	0	1	1	0
$B_3$	0	0	1	0	0	1
$B_4$	1	0	0	0	0	0
$B$	1	0	0	1	1	0
$SA_S$	6	2	4			
$SA_S^{-1}$	4	5	1			

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$H$	0	1	0	1	0	0	0	1	1	1	0							
$\tau$	0	0	1	0	0	1	0	1	1	0	0	1	0	1	1	0	1	1

Slika 15: Primer komponent kompaktne priponskega drevesa nad besedo »KOKOŠ\$«.

števila zapisana z  $\log n$  biti ( $n = 6$ ). Ekvivalentno priponsko drevo potrebuje vsaj 18B (oziroma 142 bitov), priponsko polje pa potrebuje vsaj 9 B (oziroma 36 bitov, pri čemer priponsko polje, ki simulira priponsko drevo, potrebuje približno 12 B oziroma 90 bitov). Pri tem nobena podatkovna struktura ne vključuje dodatnih podatkovnih struktur, ki omogočajo operacije v konstantnem času. Nekompaktne podatkovne strukture po navadi ne uporabljajo zapisa celih števil z  $O(\log n)$  biti, kot je bilo uporabljeno v teh izračunih, ampak se uporablja vsaj 1 B (8 bitov), po navadi 4 B, kar pa dodatno poveča razliko.

**Izboljšave.** Prostorsko zahtevnost kompaktne priponskega drevesa je mogoče še dodatno znižati na  $|CSA| + o(n)$ . Russo idr. [18] so predstavili način, kako doseči to prostorsko zahtevnost. To so dosegli tako, da so vzorčili  $O(n/\delta)$  vozlišč, pri čemer je  $\delta = \omega(\log_{|\Sigma|} n)$  in predstavlja faktor vzorčenja. Vzorčeno drevo potrebuje  $o(n) = O(n/\log_{|\Sigma|} n)$  bitov. Poleg vzorčenja topologije drevesa so opustili tudi  $LCP$  polje. To jim je omogočilo znižanje prostorske zahtevnosti iz  $|CSA| + 6n + o(n)$  na  $|CSA| + o(n)$  bitov. Na ta račun pa se je dvignila časovna zahtevnost nekaterih operacij za faktor  $\omega(\log n)$ , če se želi ohraniti  $o(n)$  dodatnega prostora. Nekatere od teh operacij so predhodno potrebovale konstanten čas izvajanja.

Preden nadaljujemo z gradnjo in poizvedbami nad  $CST$ , pogledjmo si primer človeškega genoma iz začetka poglavja. S priponskim drevesom je potrebnih 144 GB notranjega pomnilnika za indeksiranje celotnega človeškega genoma. Z uporabo kompaktnega priponskega drevesa, namesto priponskega drevesa, pa je potrebnih približno 3 GB delovnega pomnilnika za celotnega genoma. Razlika v zahtevanem prostoru omogoča, da se lahko celotno kompaktno priponsko drevo shrani v delovnem spominu, v nasprotju s priponskim drevesom, za katerega to ni mogoče<sup>1</sup>.

## 5.1 POIZVEDBE

Kompaktna predstavitev priponskega drevesa je ekvivalentna priponskemu drevesu, zato mora podpirati iste poizvedbe nad besedo  $T$ , kot jih podpira priponsko drevo. Najpreprostejša poizvedba je  $prisotnost(T, P)$ . V kompaktnem priponskem drevesu se prisotnost vzorca  $P$  išče s pomočjo vzvratnega iskanja (angl. *Backward Search*) vzorca v priponskem polju  $SA$ . Vzratno iskanje za predhodno predstavljeno kompaktno priponsko polje je prikazano v Algoritmu 6 in potrebuje  $O(mt_\Psi)$  časa, da se izvrši. Za drugačno implementacijo kompaktnega priponskega polja se lahko nadomesti vrstico 6 v Algoritmu 6 z binarnim iskanjem nad  $\Psi_c$ , za kar je potrebno  $O(m \log nt_\Psi)$  časa. Sicer se lahko uporabi kakšen drug, bolj učinkovit algoritem za iskanje vzorca. Algoritem 6 vrne interval v priponskem polju, zato je treba preveriti, ali je  $[s, e] \neq [-1, -1]$ , za kar je potreben konstanten čas. Torej je potrebno  $O(mt_\Psi)$  časa za preverjanje prisotnosti vzorca ali  $O(m \log nt_\Psi)$  z uporabo binarnega iskanja. V primeru, da se uporabi predhodno predstavljeno kompaktno priponsko polje, pa se poizvedba izvrši v času  $O(m)$ .

Algoritem vzvratnega iskanja, ki je predstavljen v Algoritmu 6, preveri prisotnost vzorca v priponskem polju tako, da se v koraku  $k$  najde interval pripon  $[s, e]$ , ki se začne z znakom  $P[k]$  in se nadaljuje z nizom  $P[k + 1 : m]$ . Prvi interval  $[s, e]$  predstavlja vse pripone, ki se začnejo z znakom  $P[m]$  [6]. Na primer, če želimo preveriti  $prisotnost(\text{»KOKOŠ$«, »KO«})$ , lahko to storimo s kompaktnim priponskim drevesom, prikazanim na Sliki 15. Začetni interval  $[s, e] = [4, 5]$ , pri čemer je  $C[P[m]] = C[O]$  oziroma  $C[3]$ , saj je  $S[3] = O$ . V edinem koraku zanke je interval  $[s', e']$  enak  $[1, 2]$ , saj je  $c = K$  oziroma  $c = 2$ , saj je  $S[2] = K$ . Vrednost  $s'$  se izračuna kot  $rang_1(B_2, 3) + 1$  in z uporabo Slike 15 izračunamo, da je  $s' = 0 + 1 = 1$ . Podobno lahko preverimo tudi vrednost  $e' = rang_1(B_2, 5) = 2$ . Zadnji korak pa izračuna novi interval  $[s, e] = [C[2] + 1, C[2] + 2] = [1 + 1, 1 + 2] = [2, 3]$ . Ker  $s$  ni večji od  $e$ , vzvratno iskanje vrne  $[2, 3]$ , kar pomeni, da je vzorec prisoten v besedi.

<sup>1</sup>Nekateri strežniki omogočajo večjo količino delovnega pomnilnika, ki presega 144 GB. Večina potrošniških računalnikov pa še vedno uporablja med 8 GB in 64 GB delovnega pomnilnika.



---

**Algoritem 6:** Iskanje intervala v SA (del CST-ja), v katerem je prisoten vzorec  $P$  [6]

---

**Vhod:** Kompaktno priponsko drevo  $CST$ , vzorec  $P$

**Izhod:** Del priponskega polja, ki se začne z  $P$

```

1   $(s, e) = (C[P[m]] + 1, C[P[m] + 1])$ 
2  za  $i = (m - 1)..1$ 
3      če  $s > e$  potem
4          return  $[-1, -1]$ 
5       $c = P[i]$ 
6       $(s', e') = (rang_1(B_c, s - 1) + 1, rang_1(B_c, e))$ 
7       $(s, e) = (C[c] + s', C[c] + e')$ 
8  če  $s > e$  potem
9      return  $[-1, -1]$ 
10 return  $[s, e]$ 

```

---

Naslednja poizvedba je  $številnPonovitev(T, P)$ , ki vrne število pojavov vzorca  $P$  v besedilu. V kompaktnem priponskem drevesu pa je poizvedba ponovno implementirana z uporabo vzratnega iskanja, prikazanega v Algoritmu 6. Poizvedba  $številnPonovitev(T, P)$  je implementirana kot razlika  $e - s + 1$ , kjer  $s$  predstavlja prvo pripono, ki se začne z vzorcem  $P$ , in  $e$  je zadnja taka pripona, torej je razlika  $e - s + 1$  število pripon. Torej je za poizvedbo ponovno potrebuje  $O(mt_\Psi)$  oziroma  $O(mt_\Psi \log n)$  časa.

Zadnja predstavljena poizvedba pa je  $seznamPojavov(T, P)$ , ki vrne vse indekse pojavov vzorca  $P$  v besedi  $T$ . V kompaktnem priponskem drevesu je poizvedba ponovno implementirana z uporabo vzratnega iskanja, prikazanega v Algoritmu 6. Z vzratnim iskanjem se izračuna interval v priponskem polju  $SA[s : e]$ . Za izračun položajev ponovitev vzorca v besedilu je potrebno ustvariti seznam  $[SA[s], SA[s + 1], \dots, SA[e]]$ , kar zahteva še dodatnih  $occ = e - s$  korakov, pri čemer vsak korak potrebuje  $O(t_{SA})$  časa, da se izvrši. Torej poizvedba  $seznamPojavov(T, P)$  potrebuje  $O(mt_\Psi + occ \cdot t_{SA})$  oziroma  $O(mt_\Psi \log n + occ \cdot t_{SA})$  časa. S predhodno predstavljeno implementacijo kompaktne priponskega drevesa pa je potrebno  $O(m + occ \cdot \log n)$  časa.

Iz implementacije poizvedb nad kompaktnim priponskim drevesom je razvidno, da so vse tri poizvedbe implementirane zgolj z uporabo kompaktne priponskega polja. Iz tega se lahko sklepa, da sta topologija drevesa  $\tau$  in  $LCP$  polje odvečni podatkovni strukturi. To velja zgolj za osnovne poizvedbe nad besedilom  $T$ , ki so lahko izvršene zgolj z uporabo kompaktne priponskega polja v enakem času. Poizvedbe, kot so najdaljši ponavljajoči se podniz, najdaljši palindrom in najdaljši skupni niz besed  $T_1$  in  $T_2$ , pa potrebujejo vse tri podatkovne strukture. Na primer, poizvedba najdaljši

ponavljajoči podniz vrne podniz  $T[SA[i] : SA[i] + sd(v)]$ , pri čemer je  $i$  skrajno levi list poddrevesa s korenem v notranjem vozlišču  $v$  in velja, da je  $LCP[i]$  največji element v  $LCP$  polju, torej zahteva  $O(nt_{SA})$  časa. Operacije  $sd(v)$  ni potrebno izračunati, saj je enaka  $LCP[i]$ , torej se še vedno izvede v  $O(nt_{SA})$  času, pri tem pa se ne uporabi topologijo drevesa. V primeru, da želimo najti drugi najdaljši ponavljajoč se podniz  $T[SA[j], SA[j] + sd(u)]$  v priponskem polju, pri čemer je  $j$  skrajno levi list poddrevesa s korenem v notranjem vozlišču  $u = sl(v)$ , pa potrebujemo tudi topologijo drevesa  $\tau$  za izračun priponske povezave. Ta se izračunana kot:

$$sl(v) = lca(izbira_{01}(\tau, \Psi(rang_{01}(\tau, v - 1) + 1)), izbira_{01}(\tau, \Psi(rang_{01}(\tau, zapri(\tau, v))))).$$

Za izračun drugega najdaljšega ponavljajočega se podniza je potrebno  $O(nt_{SA} + t_{\Psi})$  časa ter se uporabijo vse tri podatkovne strukture kompaktne priponskega drevesa [3, 5, 6].

## 5.2 GRADNJA

Kompaktno priponsko drevo je mogoče zgraditi iz priponskega drevesa. Tak način gradnje kompaktne priponskega drevesa ni prostorsko učinkovit, saj je potrebno najprej zgraditi celotno priponsko drevo, ki ga želimo nadomestiti s kompaktno predstavitvijo. Zato bi bilo bolje zgraditi kompaktno priponsko drevo neposredno iz besede, kot je to storjeno za priponsko drevo in priponsko polje.

Kompaktno priponsko drevo je mogoče zgraditi neposredno iz vhodne besede  $T$ . Pri tem so potrebne dodatne podatkovne strukture za izgradnjo posameznih komponent. Tudi te dodatne podatkovne strukture so kompaktne podatkovne strukture, zato je drevo zgrajeno v kompaktnem prostoru. Izgradnja kompaktne priponskega drevesa poteka v sledečih treh korakih:

1. izgradnja kompaktne priponskega polja  $SA$  iz besede  $T$ ,
2. izgradnja kompaktne predstavitve polja  $LCP$  iz kompaktne priponskega polja  $SA$  in besede  $T$ ,
3. izgradnja kompaktne predstavitve topologije drevesa  $\tau$  iz  $LCP$  polja.

**Izgradnja CSA.** Prva zgrajena podatkovna struktura je kompaktno priponsko polje  $SA$ . Obstajajo različne implementacije kompaktne priponskega polja, zato ni mogoče predstaviti splošnega algoritma za njihovo izgradnjo. V nadaljevanju bo predstavljen algoritem izgradnje predhodno predstavljeno implementacijo  $CSA$ . Ker je priponsko polje implementirano s funkcijo  $\Psi$  (shranjeno v istoimenskem polju), je potrebno izračunati polje  $\Psi$ . Funkcijo  $\Psi$  gradimo od leksikografsko najmanjše pripone do največje

pripone s pomočjo polja  $L[1, n]$ , pri tem je  $L[i] = T[SA[i] - 1]$  ter  $T[0] = T[n] = \$$ . Polje  $L$  se lahko zgradimo v manjših delih, zato se priponsko polje gradi v delih dolžine  $b$ . Za  $b = n/\log n$  je potrebnih  $o(n)$  bitov dodatnega prostora za shranjevanje dela priponskega polja in  $O(n \log n)$  časa za zgraditi kompaktno priponsko polje [6].

Priponsko polje  $SA$  ni razdeljeno na  $b$  delov, ampak je lažje najti najmanjše število predpon, ki se pojavijo na začetku največ  $b$  priponam, ter se na ta način razdeli priponsko polje  $SA$  na  $SA_k$  za  $1 \leq k \leq b$ . Sledi izračun pripon, ki sodijo v določen del priponskega polja  $SA_k$ . Pripone v tem delu se leksikografsko uredijo, najpogosteje z uporabo korenskega urejanja. V  $k$ -tem koraku je polje  $L$  zgrajeno za indekse od 1 do  $kb - 1$ . Vrednost celice  $kb + j$  izračunamo kot  $L[kb + j] = T[SA[j] - 1]$  [6].

Ko je polje  $L$  izračunano, se kompaktno predstavitev polja  $\Psi$  zapiše kot bitna polja  $B_c$  za vsak znak  $c \in \Sigma$ . Vsako bitno polje  $B_c$  se inicializira z ničlami. Če je  $L[i] = c$ , se nastavi  $B_c[i] = 1$ . To je lahko storjeno brez dodatnega prostora za polje  $L$ , saj ni potrebno zapisati črke  $c = T[A[j] - 1]$  v polju  $L$ , ampak se lahko neposredno zapiše enico v celico  $B_c[kb + j] = 1$  [6].

Vzporedno z izdelavo kompaktne predstavitve polja  $\Psi$  lahko izračunamo tudi polji vzorcev  $SA_S$  in  $SA_S^{-1}$ , saj se za izgradnjo sprotno izračunava priponsko polje  $SA$ . Torej za izgradnjo polj vzorcev ni potrebnega dodatnega časa. Čas, potreben za izgradnjo kompaktne priponskega polja  $SA$ , je  $O(n \log n)$ . Večina operacij potrebuje  $O(n)$  časa. Iskanje pripon, ki spadajo v določeno vedro, potrebuje  $O(n)$  časa za vsako vedro, torej potrebuje  $n/b \cdot O(n) = (n \log n)/n \cdot O(n) = O(n \log n)$  časa za izračun vseh pripone.

**Izgradnja LCP strukture.** Bitno polje  $H$ , ki predstavlja  $LCP$  polje, je naslednja zgrajena podatkovna struktura. Ker je  $H[j] = 1$  za  $j = PLCP[i] + 2i$ , se  $H$  izračuna direktno iz polja  $PLCP$ .

Vrednosti v polju  $PLCP$  so izračunane od 1 do  $n$ . Vrednost  $PLCP[1]$  je izračunana s štetjem zaporednih znakov, ki se ujemajo med  $T[1, n]$  in  $T[SA[SA^{-1}[1] - 1], n]$ . Ker je  $PLCP[i - 1] \leq PLCP[i] + 1$ , je vrednost  $PLCP[i]$  število skupnih zaporednih znakov med  $T[i + d, n]$  in  $T[SA[SA^{-1}[i] - 1] + d, n]$ , pri čemer je  $d = \max\{PLCP[i - 1] - 1, 0\}$ . Pri izgradnji bitnega polja  $H$  ni potrebo najprej izračunati polja  $PLCP$ , saj se lahko sprotno vpiše enico na mesto izračunane vrednosti  $PLCP[i] + 2i$  v bitnem polju  $H$  [6].

Za izgradnjo  $LCP$  polja, ki je predstavljeno kot bitno polje  $H$ , je potrebnih  $O(n)$  dostopov do priponskega polja  $SA$ . Vsak dostop do priponskega polja potrebuje  $O(t_{SA})$  časa (natančen čas je odvisen od implementacije priponskega polja), ki je za predhodno predstavljeno implementacijo  $t_{SA} = O(\log n)$ , in zato je za izgradnjo potrebno  $O(nt_{SA})$  časa ali za predhodno predstavljeno implementacijo  $O(n \log n)$ .

**Izgradnja topologije drevesa.** Zadnja zgrajena podatkovna struktura je topologija priponskega drevesa  $\tau$ . Čeprav se zdi, da ni mogoče zgraditi topologije drevesa, ne da bi se zgradilo celotno priponsko drevo, je to mogoče storiti zgolj z uporabo priponskega polja  $SA$  in  $LCP$  polja. Ideja algoritma za izgradnjo topologija priponskega drevesa  $\tau$  je podobna algoritmu Kasai idr. [14], ki je bil predhodno predstavljen v podpoglavju 4.3.

To je storjeno s prehodom po priponskem polju  $SA$  iz leve proti desni ter za vsako pripono  $i$  dodamo nov list. Nov dodan list  $i$  je novo skrajno desno vozlišče v do sedaj zgrajenem drevesu ter ima skupnega predhodnika  $v$  z  $(i - 1)$ -im listom na črkovni globini  $sd(v) = LCP[i]$ . Če je tako vozlišče  $v$  že v drevesu, potem list  $i$  postane njegov desni otrok. Sicer  $sd(v) < LCP[i]$  in obstaja vozlišče  $u$ , ki je lahko  $(i - 1)$ -vi list, za katerega velja  $sd(u) > LCP[i]$ . V tem primeru se ustvari novo vozlišče  $v'$ , ki postane desni otrok od  $v$  ter ima dva otroka, in sicer levega otroka  $u$  ter desnega otroka listi  $i$ . Na ta način se priponsko drevo zgradi zgolj iz priponskega polja  $SA$  in  $LCP$  polja. Predstavljena metoda je implementirana z uporabo sklada, ki hrani vozlišče  $v$  in črkovno globino  $sd(v)$ . Na sklad se ne shrani kazalec na vozlišče, ampak se shrani predstavitev poddrevesa s korenem v vozlišču  $v$  z zaporedjem uravnoveženih oklepajev  $P(v)$ . Vozlišče  $v$ , za katerega velja  $sd(v) \leq LCP[i]$ , se poišče s snetjem vozlišč iz sklada, dokler ne pridemo v vozlišče, za katerega pogoj drži. Tako vozlišče vedno obstaja, saj ima koren črkovno dolžino  $sd(koren()) = 0$ . Na sklad je dodanih največ  $n - 1$  vozlišč, ker ima priponsko drevo največ  $n - 1$  notranjih vozlišč. Da bo na koncu izgradnje sklad prazen in topologija drevesa  $\tau$  pravilna, se sklad izprazni in na ta način se pridobi pravilno topologijo  $\tau$ , saj vsa vozlišča na skladu ne shranjujejo topologije za njihovo skrajno desno poddrevo, ker le to še raste. Torej vsakič, ko se vozlišče sname iz sklada, se topologija poddrevesa doda staršu vozlišča, ki je vozlišče na vrhu sklada [6].

Izgradnja topologije drevesa poteka v času  $O(nt_{SA})$  oziroma  $O(n \log n)$  za predhodno predstavljeno implementacijo  $CSA$ . V vsakem koraku je v topologijo drevesa dodan nov list. Za vsak list lahko dodamo novo vozlišče na sklad, če za črkovno dolžino vozlišča  $v$ , ki je trenutno na vrhu sklada, velja  $sd(v) < LCP[i]$ , pri čemer je  $i$  zaporedno število pripone, ki jo predstavlja list v priponskem polju. Za vsak list so iz sklada odvzeta vozlišča, za katera velja  $sd(v) \geq LCP[i]$ . V času izgradnje drevesa je na sklad potisnjenih in snetih največ  $n - 1$  vozlišč. Ker postopek ustvari  $n$  listov in največ  $n - 1$  notranjih vozlišč, je  $O(nt_{SA})$  oziroma  $O(n \log n)$  potreben čas izgradnje topologije drevesa  $\tau$ .

**Izrek 5.9.** Časovna zahtevnost izgradnje kompaktnega priponskega drevesa za vhodno besedo  $T$  dolžine  $n$  je  $O(n \log n)$  ter je v času izgradnje prostorska zahtevnost vedno kompaktna.

*Dokaz.* Kompaktno priponsko drevo je sestavljeno iz treh delov, torej je potrebno analizirati časovno zahtevnost izgradnje vsakega dela. Za izgradnjo kompaktne priponskega polja je potrebno  $O(n \log n)$  časa, saj je treba najti pripone, ki so del posamičnega vedra priponskega polja.

Za izgradnjo kompaktne različice  $LCP$  polja je tudi potrebno  $O(n \log n)$  časa, saj je za vsako pripono potrebno izračunati dolžino predpone, v kateri se ujema s predhodno pripono. Pri tem je potreben za vsako pripono en dostop do priponskega polja, ki traja  $O(\log n)$  časa za predstavljeno implementacijo  $CSA$ , torej je za  $n$  pripon potrebno  $O(n \log n)$  časa.

Za izgradnjo topologije drevesa pa je tudi potrebno  $O(n \log n)$  časa, saj je za vsako pripono potrebo ustvariti nov list ter novo notranje vozlišče, če ne obstaja vozlišče na poti do skrajno desnega lista, za katerega velja  $sd(v) = LCP[i]$ . V vsakem koraku je potrebno izračunati  $LCP[i]$ , ki potrebuje  $O(\log n)$  časa.

Skupni čas za izgradnjo kompaktne priponskega drevesa iz besede  $T$  je torej  $O(n \log n) + O(n \log n) + O(n \log n) = O(n \log n)$ .  $\square$

## 6 EMPIRIČNA EVALVACIJA

Do sedaj so bile predstavljene različne implementacije indeksov besed. V tem poglavju so ti indeksi med seboj empirično primerjani. Testirajo se naslednje podatkovne strukture:

1. priponsko drevo,
2. kompaktno priponsko drevo,
3. priponsko polje in
4. priponsko polje z LCP poljem.

Za vsako od naštetih podatkovnih struktur so izmerjene tri vrednosti: velikost podatkovne strukture med testiranjem, čas gradnje in čas za poizvedbe v podatkovni strukturi.

To poglavje je razdeljeno na tri dele. Najprej bomo predstavili testno okolje, metodo testiranja in testne besede. Nato bodo predstavljeni rezultati testov. V zadnjem delu poglavja pa bo predstavljena razprava rezultatov primerjav.

### 6.1 EKSPERIMENTALNO OKOLJE

Preden se lahko izdela primerjavo podatkovnih struktur je potrebno predstaviti okolje, v katerem bo izdelana primerjava, metodo, ki bo uporabljena za izdelavo primerjave, ter same testne primere, ki bodo uporabljeni za primerjavo. V tem podpoglavju bodo najprej predstavljeni testni primeri, nato bo predstavljena uporabljena testna metoda ter na koncu bo predstavljen stroj in operacijski sistem, ki sta bila uporabljena za testiranje.

#### 6.1.1 Testni primeri

Empirična primerjava je narejena nad dvema vhodnima besedama, ki sta prikazani v Tabeli 4. Prva testirana beseda je daljše zaporedje DNK [9], ki je zlepek različnih zaporedij DNK in je označena kot DNKSEKVENCA. Izbrana je bila kot primer vhodne besede, ki se uporablja v bioinformatiki. Ta vhodna beseda je bila predhodno uporabljena za testiranje implementacije kompaktne priponskega drevesa od Välimäki idr. [3]. Abeceda  $\Sigma$  besede DNKSEKVENCA je sestavljena iz nukleotidov, ki jih

označimo s črkami A, C, T in G. Zaradi lažje implementacije smo se odločili, da nukleotide predstavimo kot 7-bitne ASCII znake, namesto 3-bitnega kodiranja (dodatni bit je potreben za predstavitev konca besede).

Tabela 4: Primerjava besedil, ki smo jih uporabili za primerjavo različnih indeksov besed

Ime testne besede, Oznaka besede	Število znakov	Velikost abecede	Velikost na disku [MB]
Ivan Cankar, Na klancu [10], NAKLANCU	317803	53	25,851
zaporedje DNK [9], DNKSEKVENCA	52428800	5	26,767

Druga testirana beseda je Cankarjev kratki roman Na klancu [10], ki je označen kot NAKLANCU. Roman je bil izbran, saj predstavlja tipično vhodno besedo v naravnem jeziku, ki je v tem primeru slovenščina. Abeceda  $\Sigma$  besede NAKLANCU sestoji iz črk slovenske abecede (velikih in malih), presledkov in ločil. Ker smo se odločili, da predstavimo DNKSEKVENCA z ASCII znaki, je bilo potrebno NAKLANCU pretvoriti v 7-bitne ASCII znake. Zato so bili vsi ne-ASCII znaki odstranjeni oziroma zamenjani z njihovimi ASCII alternativami. Na primer znak, ki predstavlja »...«, je bil zamenjan s tremi pikami, vse črke z naglasi (kot so strešice, ostrivci ter drugi) so bile zamenjane z osnovnim znakom, zato š postane s in é postane e. Odstranjene so bile tudi vse prazne vrstice ter ločila poglavij, ki so bila označena z »\*\*\*«, saj le-ta ne vplivajo na vsebino besedila. Odstranjeni so bili tudi vsi simboli, ki niso vidni bralcu, prisotni v besedilu. Na ta način se je začetna dolžina besedila znižala iz 319843 znakov na 317803.

**Podaljševanje besedila.** V drugem stolpcu Tabele 4 so prikazane dolžine besed, ki se uporabljajo za testiranje. Pri tem opazimo, da je beseda NAKLANCU krajša od besede DNKSEKVENCA. Zato jo je potrebno podaljšati. Ker je malo verjetno, da se vhodna beseda ponovi  $k$ -krat, dokler ne doseže primerne velikosti, predvsem v naravnem jeziku, smo uporabili naprednejša metoda podaljševanja besedila. Metoda vzame manjše podnize besedila ter naredi stik med začetno besedo in podnizom. Tako dobljena beseda je bolj verjetna, saj je večja verjetnost, da se manjši deli besede ponovijo, za razliko od celotne besede. Predlagana metoda podaljševanja besede je prikazana v Algoritmu 7. Pri tem metoda predpostavi, da je beseda dolga vsaj 3000 znakov. Ta metoda je primerna za podaljševanje daljših besed, sicer pa je mogoče v metodi spremeniti parameter  $i$  in tako prilagoditi metodo drugim besedam.

Predlagana metoda podaljša besedo na velikost  $s_{max}$ . Ta velikost je lahko dolžina najdaljše besede ali pa je poljubna vrednost, bodisi manjša od dolžine največje besede

**Algoritem 7:** Metoda podaljševanja vhodne besede**Vhod:** Vhodna beseda  $T$ , velikost  $s_{max}$ **Izhod:** Besedilo  $T_0$ 


---

```

1  $T_0 \leftarrow T$ 
2  $i \leftarrow 500$ 
3 while  $|T_0| < s_{max}$  do
4    $T_0 \leftarrow T_0 \cdot T[i : 6i]$ 
5    $i \leftarrow i + 500$ 
6   while  $6i > |T|$  do
7      $i = i/4$ 
8 vrni  $T_0[1 : s_{max}]$ 

```

---

bodisi večja. Če je beseda daljša od velikosti  $s_{max} < |T|$ , bo predlagana metoda skrajšala besedo na  $|T_0| = s_{max}$ . Za potrebe testiranja smo se odločili, da je  $s_{max} = 25000000$ , kar je več kot zadostna dolžina.

### 6.1.2 Metoda testiranja

Metoda testiranja je prikazana s psevdokodo na Algoritmu 8. Najprej se izgradi podatkovna struktura  $PS$  ter se izmeri čas gradnje. To je v psevdokodi prikazano v vrstici 5. Nato pa se izmeri še čas, ki je potreben za poizvedbo  $prisotnost(T, P)$  s pomočjo podatkovne strukture  $PS$ , prikazano v vrstici 7 Algoritma 8. Poizvedba  $prisotnost(T, P)$  je izbrana, saj je v biologiji pogosto vprašanje prisotnost gena (vzorcev  $P$ ) v zaporedju DNK (vhodna beseda  $T$ ), ne pa natančen položaj tega gena ali število ponovitev gena v zaporedju DNK. Čas poizvedbe se meri za vzorce velikosti 5, 50, 500 in  $\log n$  znakov. Preden se podatkovna struktura  $PS$  izbriše iz pomnilnika, se rezultati testiranja shranijo v seznam, kot je prikazano v vrstici 10 Algoritma 8. Da se zmanjša vpliv drugih procesov na rezultate testiranja, se vsak test zažene petkrat.

Poleg testiranja čas gradnje podatkovnih struktur in poizvedb v njih želimo izmeriti tudi njihovo velikost. Velikost podatkovne strukture se ne da izmeriti brez uporabe pomnilniškim profilerjem (angl. *memory profiler*). Le-ta bo naknadno dodana k rezultatom ostalih meritev. Pri tem bomo merili največji zasedeni prostor v času testiranja podatkovne strukture  $PS$  za besedo dolžine  $n$ .

### 6.1.3 Stroj in operacijski sistem

Za izdelavo primerjav podatkovnih struktur uporabljamo računalnik s procesorjem Intel Core i3 5005U z dvema jedroma in štirimi nitmi ter s taktom 1,9 GHz. Računalnik ima 4 GB delovnega pomnilnika, od katerega je ob zagonu zasedenih 1,35 GB z operacij-



**Algoritem 8:** Pseudokoda primerjave indeksov vhodne besede  $T$ **Vhod:** Vhodna beseda  $T$ , število znakov v najdaljšem testiranem nizu  $n_{max}$ **Izhod:** Seznam rezultatov testiranja

---

```

1   $n \leftarrow 500$ 
2  dokler  $n \leq n_{max}$ 
3      za  $PS \in \{ST, SA, SA + LCP, CST\}$ 
4          za  $i = 1, \dots, 5$ 
5               $s \leftarrow \text{čas}(); \text{Izgradi}(PS, T[1 : n - 1] \cdot \$); k \leftarrow \text{čas}()$ 
6               $t_{Izg} \leftarrow k - s$ 
7              za  $m \in \{5, 50, 500, \log n\}$ 
8                   $s \leftarrow \text{čas}(); \text{prisotnost}(PS, T[n : n + m]); k \leftarrow \text{čas}()$ 
9                   $t_m \leftarrow k - s$ 
10              $\text{Shrani}(PS, n, t_{Izg}, t_5, t_{50}, t_{500}, t_{log})$ 
11              $\text{Izbriši}(PS)$ 
12   $n \leftarrow 2 \cdot n$ 

```

---

skim sistemom in emulatorjem terminala (angl. *terminal emulator*). Poleg delovnega pomnilnika ima računalnik še 8 GB Swap prostora na trdem disku, ki je namenjen shranjevanju neuporabljenih pomnilniških strani v primeru prezasedenega delovnega pomnilnika in posledično sproščanja le-tega. Operacijski sistem računalnika je Fedora 42 in uporabljen je Linux kernel 6.14.11-300.

Primerjava smo izdelali v programskem jeziku C++<sup>1</sup>. Zato čase, potrebne za izgradnjo struktur in poizvedb v njih, merimo z beleženjem trenutnega časa pred začetkom, ki se shrani v spremenljivko **start**, in po koncu operacije, ki se shrani v spremenljivko **konec**. Meritve so implementirane s funkcijo `high_resolution_clock::now()`, ki je del standardne knjižnice programskega jezika C++ in vrne natančen čas trenutka, v katerem se je funkcija izvedla. Čas izvajanja izračunamo kot razlika med časom ob koncu izvajanja in časom ob začetku izvajanja operacije, kar se v C++ izvede s funkcijo `duration_cast<nanoseconds>(konec - start).count()`. Za izgradnjo podatkovnih struktur uporabljamo naslednje knjižnice: za priponsko drevo je uporabljena implementacija iz knjižnice [12], za kompaktno priponsko drevo je uporabljena implementacija iz knjižnice SDNL [11], za priponska polja pa je uporabljena implementacija iz knjižnice [30]. Slednja zgradi priponsko polje v  $O(n)$  času z algoritmom, ki sta ga predstavila Timoshevskaya in Feng [31]. Algoritem je izboljšava predhodno predstavljenega algoritma Ko in Aluru [25]. Uporabljamo pa različica *LCP*,  $Q - LCP$  polje, ki je bila predstavljena v podpoglavju 4.1.

---

<sup>1</sup>Koda je dostopna na povezavi <https://github.com/GioGiou/MagisterskaNalogaKoda>.

Poleg časovne zahtevnosti gradnje in poizvedb v podatkovnih strukturah nas zanima, koliko prostora zasede posamezna podatkovna struktura v delovnem pomnilniku. Zanima nas največji zaseden prostor v času testiranja posamezne podatkovne strukture. Tega merimo z uporabo pomnilniškega profilerja. Uporabljamo pomnilniški profiler Bytehound [20], saj omogoča grafični prikaz porabe pomnilnika v času izvajanja programa.

Ker je primerjava izdelana v C++, je potrebno kodo prevesti v izvršljivo datoteko, kar storimo s prevajalnikom GCC 15.1.1 s sledečim ukazom:

```
g++ -std=c++11 -O3 -DNDEBUG -I ./include -L ./lib main.cpp -o //
main -lsdsl -ldivsufsort -ldivsufsort64 -lsuffix -lsais
```

Pri prevajanju programa se uporablja nekaj zastavic, ki določajo vrednosti parametrov prevajalnika. Večina zastavic se nanaša na uvoz knjižnic v prevajalni proces, in sicer `-I ./include` nastavi pot do zaglavnih datotek (angl. *header files*), `-L ./lib` nastavi pot do strojne kode knjižnice in zastavice `-lsdsl`, `-ldivsufsort`, `-ldivsufsort64`, `-lsais` in `-lsuffix` uvozijo potrebne knjižnice za izgradnjo izvršljive datoteke. Zastavica `-O3` določa raven optimizacije izvršljive datoteke. Ker uporabljamo profiler, je potrebno pred začetkom testiranja indeksov besed zagnati tudi profiler. Test izvedemo z ukazom:

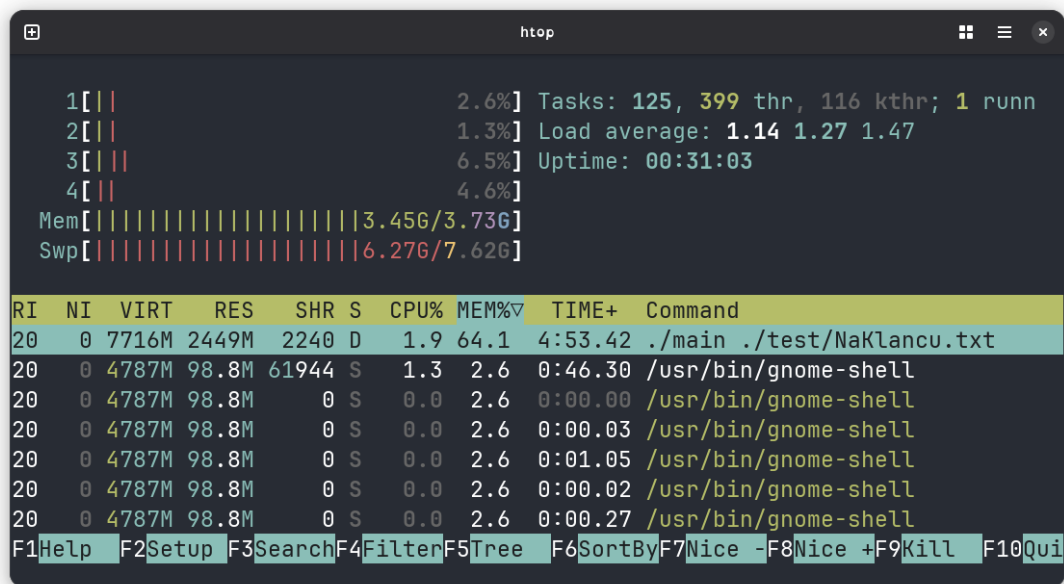
```
LD_PRELOAD=~/.bytehound/libbytehound.so ./main ./test/Test.file
```

Program poženemo ter se mu kot prvi parameter podamo ime datoteke s testnimi podatki. Ker se uporablja profiler za merjenje zasedenega prostora, se v spremenljivko okolja `LD_PRELOAD` shrani profiler kot deljeni objekt (angl. *shared object*), ki se zažene pred začetkom programa. Stopnja profiliranja pa shranimo v spremenljivki okolja `MEMORY_PROFILER_LOG` in jo nastavimo na vrednost `info`.

## 6.2 REZULTATI

V tem podpoglavju bomo predstavili rezultate testiranja različnih indeksov besed. Testiranje je bilo izdelano z metodo in računalnikom predstavljenim v podpoglavju 6.1. Meritve lahko razdelimo na dva dela, in sicer: meritvi vezani na gradnjo, ki sta čas gradnje in velikost podatkovne strukture, ter meritve časov poizvedb z uporabo indeksov. Zato bo tudi sledeče podpoglavje razdeljeno na dva dela. V vsakem delu bodo ločeno predstavljeni rezultati testiranja za vsako testno besedo iz Tabele 4.

Dolžina najdaljše vhodne besed  $n_{max}$  je bila sprva nastavljena na 4000000 znakov. Med testiranjem priponskega drevesa besede dolžine 2048000 znakov opazimo, da je čas, potreben za izgradnjo priponskega drevesa, presegal pet minut. V tem trenutku je bil delovni pomnilnik v celoti zaseden ter 6 GB pomnilniških strani je bilo premaknjenih



Slika 16: Posnetek zaslona upravljalnika opravil Htop med izgradnjo priponskega drevesa za besedilo dolžine 2048000 znakov.

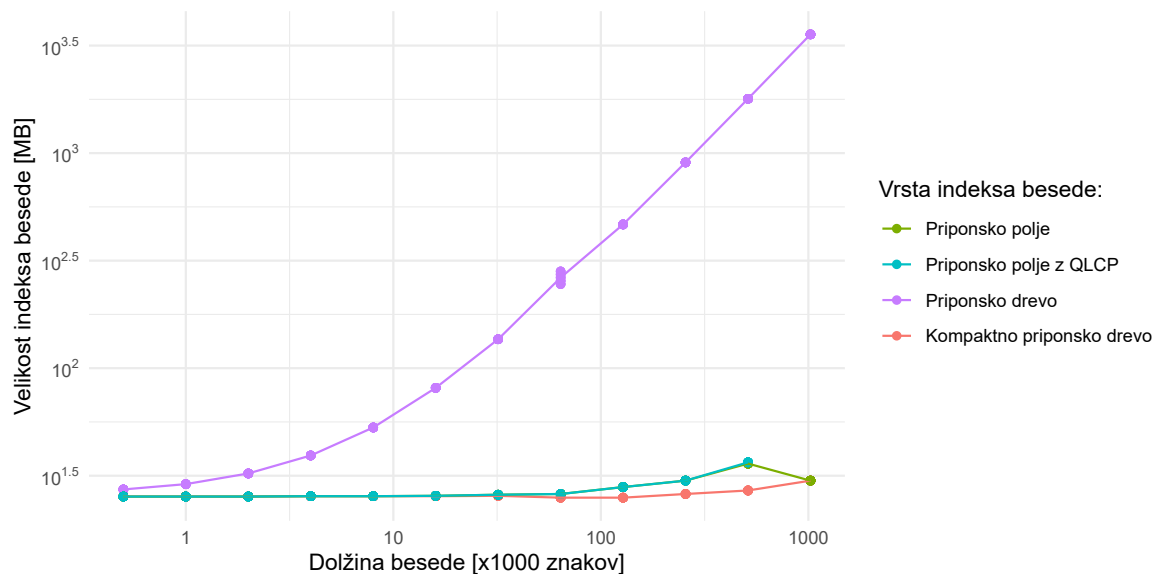
na Swap prostor. Pri tem je bil računalnik neodziven in proces je bil v neprekinjenem spanju (angl. *Uninterruptible sleep* ali stanje D). Na Sliki 16 je prikazan upravljalnik opravil Htop v času izgradnje priponskega drevesa za besedo dolžine 2048000 znakov. Proces v modri vrstici predstavlja program za testiranje časa izgradnje in poizvedbo ter prostorske zahtevnosti indeksov besed. V stolpcu, označenem S (Stanje ali angl. *Status*), je vidno, da je stanje procesa označeno kot D. Po več kot petih minutah od začetka gradnje priponskega drevesa za besedo dolžine 2048000 znakov smo se odločili, da se proces ustavi. Posledično se je znižala velikost najdaljše testirane besede na 1024000 znakov. Med testiranjem smo tudi opazili, da se *LCP* polje ne zgradi za besede dolžine 1024000 znakov in smo se odločili, da se ta test v zadnjem koraku izpusti.

### 6.2.1 Gradnja

Najprej bodo predstavljeni rezultati gradnje. Prvo bodo predstavljeni rezultati meritev velikosti podatkovnih struktur, zatem pa bodo predstavljeni rezultati časov potrebnih za gradnjo.

**Velikost podatkovne strukture.** Rezultati meritve prostorske zahtevnosti so prikazani na Sliki 17 za DNKSEKVENCA in na Sliki 18 za NAKLANCU. Iz obeh testiranj je vidno, da je prostorska zahtevnost priponskega drevesa večja kot prostorska zahtev-

nost ostalih indeksov. Pri tem vidimo tudi, da prostorska zahtevnost raste z velikostjo besede, za vse 4 testirane podatkovne strukture.



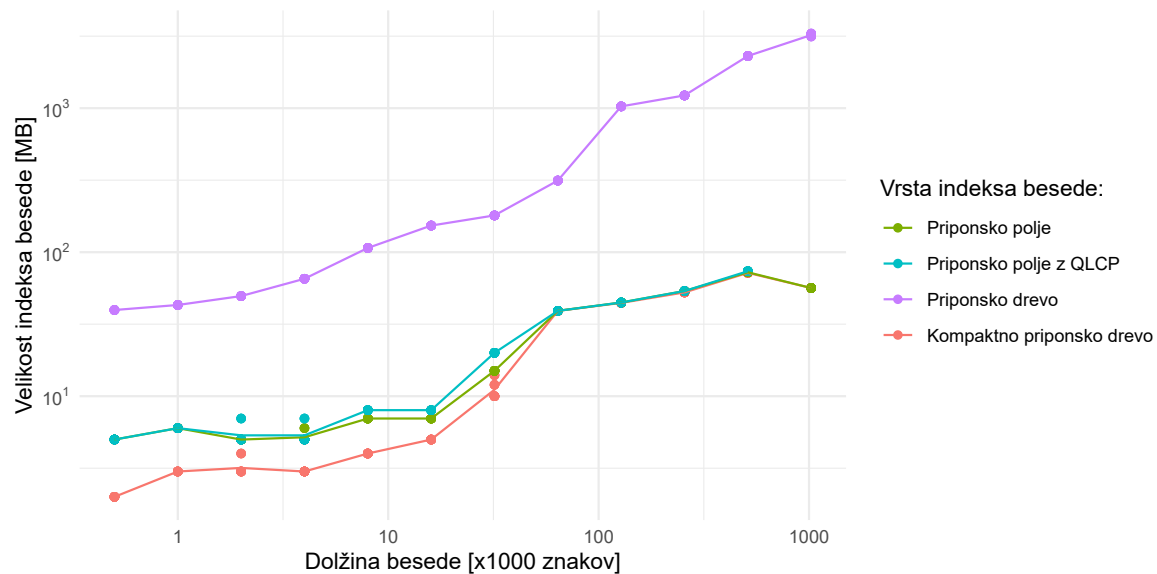
Slika 17: Graf velikosti indeksov izgrajenih iz besed različnih velikosti. Vhodna beseda je DNKSEKVENCA.

Iz rezultatov testiranja za DNKSEKVENCA, Slika 17, vidimo, da priponsko polje (označeno z zeleno barvo), priponsko polje z LCP poljem (označenim z modro barvo) in kompaktno priponsko drevo (označeno z rdečo barvo) potrebujejo bistveno manj prostora, kot ga potrebuje priponsko drevo (označeno z vijolično barvo).

Podobno se lahko vidi tudi za NAKLANCU, Slika 18. Velikost vseh podatkovnih struktur linearno raste z dolžino vhodne besede, pri čemer priponsko drevo (označeno z vijolično barvo) potrebuje bistveno več prostora kot ostale podatkovne strukture. Pri tem pa se vidi, da je razlika v zasedenem prostoru med priponskim drevesom in ostalimi indeksi nižja in približno konstantna skozi celotno izvajanje testa, za razliko od testa nad DNKSEKVENCA.

**Čas potreben za gradnjo.** Rezultati so prikazani na Sliki 19 za DNKSEKVENCA ter na Sliki 20 za NAKLANCU. V obeh primerih vidimo, da čas gradnje indeksov raste linearno z dolžino besede. V obeh testnih primerih se priponsko polje (označeno z zeleno barvo) zgradi najhitreje. Iz slik se zdi, da kompaktno priponsko drevo (označeno z rdečo barvo) potrebuje konstanten čas za gradnjo, vendar se iz rezultatov testiranja vidi majhno rast, ki ni vidna na slikah.

Časi, izmerjeni pri testiranju z DNKSEKVENCA, prikazani na Sliki 19, za indeksa priponsko drevo (označeno z vijolično barvo) in priponsko polje z dodanim LCP poljem (označen z modro barvo) so približno stokrat daljši od časov gradnje priponskih polj (označenih z zeleno barvo). Opazi se tudi, da test priponskih polji potrebuje manj



Slika 18: Graf velikosti indeksov izgrajenih iz besed različnih velikosti. Vhodna beseda je roman NAKLANCU.

kot milisekundo za zgraditi priponska polja za besede krajše od vključno 4000 znakov oziroma prve štiri dolžine besed.

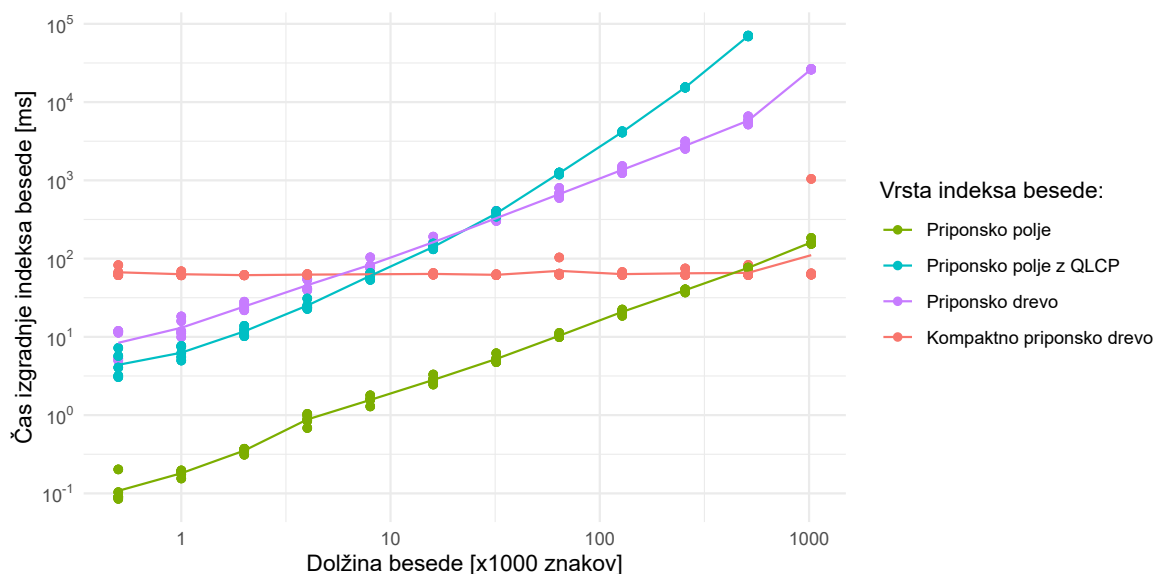
Podobni rezultati so izmerjeni tudi za NAKLANCU, prikazani na Sliki 20. Priponsko drevo (označeno z vijolično barvo) in priponsko polje z dodanim LCP poljem (označeno z modro barvo) potrebuje približno stokrat daljši čas kot priponsko polje (označeno z zeleno barvo) za izgradnjo. Opazi se tudi, da test priponskih polji potrebuje manj kot milisekundo za zgraditi priponska polja za besede krajše od vključno 4000 znakov oziroma prve štiri dolžine besed.

## 6.2.2 Poizvedbe

Rezultati testiranja so prikazani na Sliki 21 za poizvedbe v DNKSEKVENCA ter na Sliki 22 za poizvedbe v NAKLANCU. Na obeh slikah so prikazani grafi rezultatov poizvedb za vzorce dolžine 5 znakov (prvi graf), vzorce dolžine 50 znakov (drugi graf), vzorce dolžine 500 znakov (tretji graf) in vzorce dolžine  $\log n$  znakov (četrti graf).

**Čas potreben za poizvedbe z vzorci dolžine 5.** Pri DNKSEKVENCA, katere rezultati testiranja so prikazani na prvem grafu Slike 21, je iskanje v kompaktnem priponskem drevesu (označeno z rdečo barvo) najhitrejše. Pri priponskih drevesih (označenih z vijolično barvo) in kompaktnih priponskih drevesih je čas iskanja neodvisen od dolžine vhodne besede. Časi iskanja s priponskimi polji (označenimi z zeleno barvo in z modro barvo) pa se povečujeta za daljše besede.

Na prvem grafu Slike 22, ki prikazuje rezultate za NAKLANCU, so še bolj vidne raz-



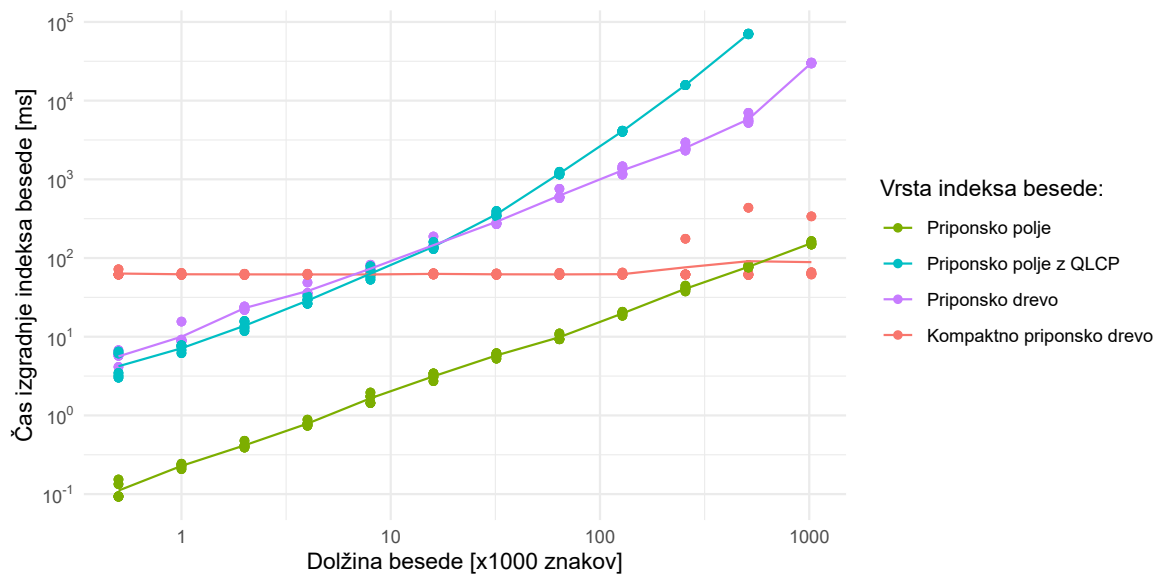
Slika 19: Graf prikazuje čas gradnje indeksov besede za različne dolžine vhodnih besed. Vhodna beseda je DNKSEKVENCA.

like med implementacijami priponskih dreves ter med implementacijami s priponskimi polji. Pri iskanju vzorca v besedi dolžine 1024000 z uporabo priponskega drevesa se vidi skok v času, ki je potreben za izvedbo poizvedbe, saj so nekatere strani pomnilnika morale biti ponovno naložene v pomnilnik iz Swap prostora.

**Čas potreben za poizvedbe z vzorci dolžine 50.** Ko se velikost vzorca poveča na 50 znakov, se čas poizvedbe za DNKSEKVENCA, prikazani na drugem grafu Slike 21, loči na dva dela, in sicer na hitrejšo iskanje s priponskimi drevesi (označenimi z vijolično barvo) in kompaktnimi priponskimi drevesi (označenimi z rdečo barvo) ter na počasnejše iskanje s priponskimi polji (označenimi z zeleno barvo in z modro barvo). Pri tem se opazi tudi premik strani iz Swap prostora na notranji pomnilnik pri testiranju priponskega drevesa besede dolžine 1024000.

Rezultati testiranja NAKLANCU, prikazani na drugem grafu Slike 22, so podobni rezultatom testiranja DNKSEKVENCA. Vidno je, da je iskanje s priponskimi drevesi oziroma kompaktnimi priponskimi drevesi desetkrat hitrejšo kot s priponskimi polji. Opazi se lahko tudi vpliv premika strani, kot se je opazilo pri drugih že predstavljenih testih za priponska drevesa besed dolžine 1024000 znakov.

**Čas potreben za poizvedbe z vzorci dolžine 500.** Če nadaljujemo na vzorce dolžine 500 znakov, so rezultati za DNKSEKVENCA, prikazani na tretjem grafu Slike 21, podobni rezultatom iskanja vzorcev dolžine 50 znakov. Pri tem je iskanje najhitrejšo v priponskih drevesih (označenimi z vijolično barvo), razen za daljše besede, ki ne morejo biti shranjene v celoti na delovnem pomnilniku. Pri tem pa se ne vidi tako



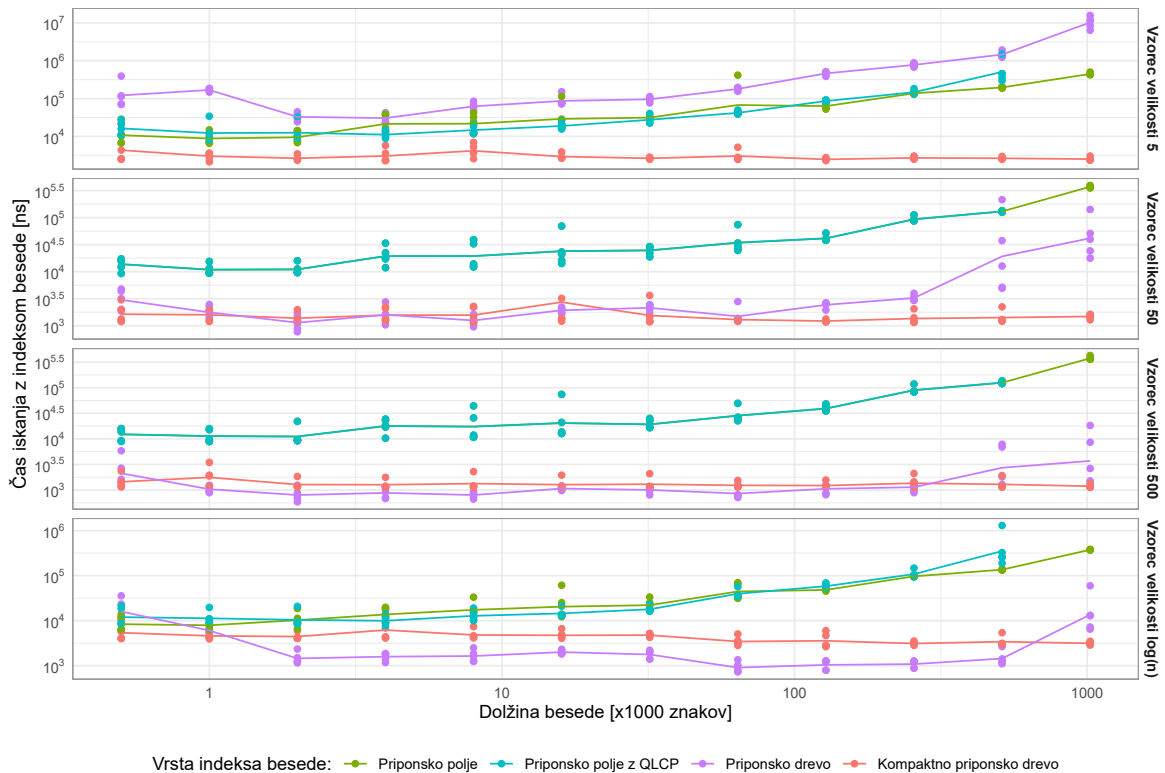
Slika 20: Graf prikazuje čas gradnje indeksov besede za različne dolžine vhodnih besed. Vhodna beseda je NAKLANCU.

drastičnega poslabšanja kot pri vzorcih dolžine 50 znakov.

Rezultati testiranja NAKLANCU so prikazani na tretjem grafu Slike 22 (tretji graf). Opazimo lahko, da so približno enaki kot pri DNKSEKVENCA in za vzorce dolžine 50 znakov. Za razliko od DNKSEKVENCA se v tem primeru izrecno vidi vpliv časa, potrebnega za zamenjavo strani v notranjem pomnilniku, na iskanje s priponskim drevesom (označenim z vijolično barvo). Vidi se tudi, da je priponsko drevo za vse teste razen za zadnja dva najhitrejši način iskanja vzorcev v besedah.

**Čas potreben za poizvedbe z vzorci dolžine  $\log n$ .** Zadnja izvedena testa sta prisotnost vzorcev dolžine  $\log n$  s pomočjo indeksov. Za DNKSEKVENCA, prikazani na četrtem grafu Slike 21, se vidi, da za krajše besede (besede, ki niso daljše od 1000 znakov) vse štiri podatkovne strukture potrebujejo približno isto časa. Za daljše besede pa je priponsko drevo (označeno z vijolično barvo) najhitrejši način iskanja. To velja zgolj dokler drevo ne preraste velikost notranjega pomnilnika in ne potrebuje zamenjave strani, kar negativno vpliva na rezultat iskanja. To se zgodi za besede dolžine 1024000 znakov.

Za NAKLANCU, katerega rezultati so prikazani na četrtem grafu Slike 22, pa so rezultati podobni. Pri tem je priponsko drevo (označeno z vijolično barvo), konstantno najhitrejši način iskanja. To velja zgolj dokler drevo ne preraste velikost notranjega pomnilnika in ne potrebuje zamenjave strani, kar negativno vpliva na rezultat iskanja. To se zgodi za besede dolžine 1024000 znakov in je v tem testu jasnejše vidno kot pri DNKSEKVENCA.



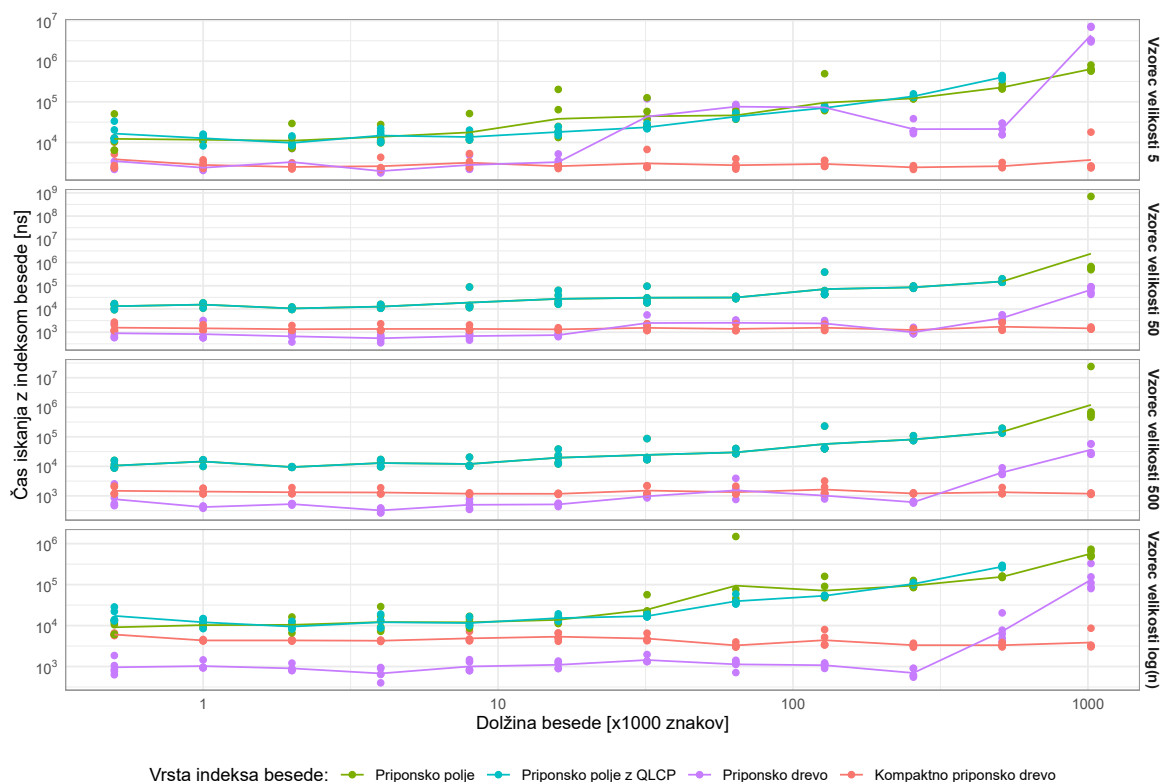
Slika 21: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih indeksih besede. Vhodna beseda je DNKSEKVENCA.

## 6.3 RAZPRAVA

Iz rezultatov testiranja opazimo, da največje priponsko drevo zasede 3,56 GB pomnilnika za DNKSEKVENCA, ki je prikazan z viola barvo na Sliki 17, ter 3,27 GB pomnilnika za NAKLANCU, ki je prikazan z viola barvo na Sliki 18. Čeprav priponsko drevo potrebuje manj pomnilnika, kot ima na razpolago sistem (4 GB), ga operacijski sistem zasede 1,3 GB, torej je potrebno nekatere strani shraniti na Swap prostoru. Premalo pomnilnika je tudi razlog za omejitev testov na besede dolžine 1024000 znakov, saj je računalnik potreboval več kot 5 minut za gradnjo enega priponskega drevesa. Ti rezultati so v skladu z razliko v hitrosti dostopa do podatkov med notranjim pomnilnikom in zunanjim pomnilnikom, ki je v tem primeru trdi disk. Trdi disk potrebuje 7-krat več časa za dostop do zaporedno zapisanih podatkov, kar se zgodi pri ne popolnoma zasedenem Swap prostoru med testiranjem priponskega drevesa za besedo dolžine 1024000. Meritev hitrosti branja in pisanja med različnimi tipi pomnilnika je izmeril tudi Jacobs [21], ki je prišel do istih izmerjenih rezultatov. Ostali indeksi zasedejo približno enak prostor. Razlog za to je težko razlikovanje prostora dveh podatkovnih struktur s pomočjo uporabljenega pomnilniškega profilerja.

Naslednje vprašanje, na katero želimo odgovoriti, je vpliv tipa abecede vhodne





Slika 22: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih indeksih besede. Vhodna beseda je NAKLANCU.

besede. Iz samega testiranja ni razvidnih bistvenih razlik pri času, potrebnem za gradnjo indeksov ter zasedenem prostoru. Podobno ni vidna nobena bistvena razlika v rezultatih iskanja vzorcev. Torej tip abecede vhodne besede ne vpliva na velikost in hitrost iskanja in gradnje podatkovnih struktur.

Zadnje vprašanje, na katerega želimo odgovoriti, je, kdaj se uporabi posamezna struktura. Prvo stvar, ki jo opazimo, je približno enaka časovna zahtevnost poizvedb med priponskim poljem brez in z dodatno LCP strukturo. Najbolj verjeten razlog je predčasna prekinitev razpolavljanja, kar zniža čas poizvedbe. Opazi se tudi dodatni čas, potreben za izgradnjo LCP polja. Najbolj verjeten razlog za veliko časovno razliko so neučinkoviti rekurzivni klici. Če pa se vrnemo nazaj na vprašanje, kdaj se uporabi posamični indeks, lahko ločimo tri načine uporabe.

**Nizko število poizvedb, ki ne omogočajo amortizacije izgradnje.** Najbolj primerna struktura za tak tip uporabe je priponsko polje, saj se ga najhitreje izgradi (Sliki 19 in 20) ter zasede malo prostora. Pri tem se iskanje lahko pospeši z uporabo LCP polja.

**Visoko število poizvedb, ki omogočajo amortizacije izgradnje, pri čemer je indeks lahko v celoti shranjen v delovnem pomnilniku.** Za tak tip uporabe je

najbolj primerna podatkovna struktura priponsko drevo, saj je od testiranih indeksov najhitrejši za iskanje vzorcev v besedi, če nismo prostorsko omejeni.

**Visoko število poizvedb, ki omogočajo amortizacije izgradnje, pri čemer indeks ni v celoti shranjen v delovnem pomnilniku.** Ko pa smo prostorsko omejeni, pa uporabimo kompaktno priponsko drevo, saj potrebuje malo prostora, čas izgradnje je kratek ter je drugi najhitrejši indeks za iskanje vzorcev v besedi.

Dobljene rezultate lahko primerjamo tudi z rezultati od Välimäki idr. [3]. Naši rezultati sovpadajo z rezultati, ki so jih dobili za DNKSEKVENCA. Pri tem pa ne moremo primerjati rezultate za naravni jezik, saj smo uporabljali kot vhodno besedo besedilo v slovenščini, oni pa so uporabljali besedilo v angleščini.

## 7 ZAKLJUČEK

Namen magistrske naloge je bil predstavitev podatkovne strukture kompaktno priponsko drevo ter primerjava le-te s priponskim drevesom in priponskim poljem. Podatkovne strukture so bile med seboj empirično primerjane.

Najprej smo predstavili vsako podatkovno strukturo. Predstavitev ni vsebovala samo predstavitev implementacije podatkovne strukture, ampak tudi predstavitev algoritmov za izgradnjo podatkovne strukture ter implementacije poizvedb nad besedo s strukturo.

Glavna razlika med priponskim drevesom, priponskim poljem in kompaktnim priponskim drevesom je v časovni zahtevnosti izgradnje ter prostorski zahtevnosti. Priponsko drevo za besedo  $T$  dolžine  $n$  potrebuje  $O(n)$  povezav, kar pomeni, da potrebuje  $O(n \log n)$  bitov ali  $O(nw)$  bitov, če se uporabljajo sistemski naslovi. Podobno priponsko polje, ki mu lahko dodamo  $LCP$  polje za pospešitev poizvedb, potrebuje  $O(n)$  celih števil, ki so shranjena z  $O(n \log n)$  biti ali  $O(nw)$  biti. Kompaktno priponsko drevo pa potrebuje  $|CSA| + 6n + o(n)$  bitov, kar v obeh predstavljanih implementacijah kompaktne priponskega polja ohrani prostorsko zahtevnost  $O(n)$  bitov. Časovna zahtevnost izgradnje priponskega drevesa in priponskega polja je  $O(n)$ . Ta se poslabša na  $O(n \log n)$  časa za kompaktna priponska drevesa, kar omogoča, da je celoten postopek izgradnje kompaktne priponskega drevesa v celoti storjen s kompaktnimi podatkovnimi strukturami.

Z empirično primerjavo so bile potrjene razlike v prostorski in časovni zahtevnosti operacij. Empirična primerjava je bila opravljena nad zaporedjem DNK in nad besedilom v naravnem jeziku (slovenščina). Empirična primerjava je merila časovno zahtevnost poizvedbe  $prisotnost(T, P)$  nad besedili različnih velikosti in vzorcev različnih dolžin. Merila je tudi časovno zahtevnost gradnje podatkovnih struktur različnih velikosti in prostorsko zahtevnost le-teh. Z empirično primerjavo se lahko potrdijo teoretične časovne razlike gradnje in poizvedb med priponskimi drevesi, priponskimi polji in kompaktnimi priponskimi drevesi. Pri tem pa se opazi tudi razlika med časovno zahtevnostjo gradnje in poizvedb, če je priponsko drevo v celoti shranjeno v delovnem pomnilniku in če je del drevesa shranjen na Swap prostoru. Te razlike ni mogoče opaziti na priponskih poljih in kompaktnih priponskih drevesih, saj je dolžina besedila prekratka, da bi jih bilo potrebno shraniti na Swap prostoru.

Rezultati testiranja so bili izdelani na računalniku z zgolj 4 GB delovnega pomnilnika in 8 GB Swap prostora, kar je relativno malo spomina, saj imajo novi osebni

računalniki vsaj 16 GB delovnega pomnilnika in procesorji za strežnike podpirajo več 100 GB delovnega pomnilnika. Torej se pojavi vprašanje, ali so kompaktne podatkovne strukture sploh še potrebne, saj imajo trenutni računalniki na voljo dovolj delovnega pomnilnika za shranjevanje celotnega priponskega drevesa. Menimo, da so še vedno potrebne, saj omogočajo shranjevanje in iskanje po večjem številu priponskih dreves hkrati. Iskanje vzorcev v večjem številu priponskih dreves naenkrat je mogoče, saj večina procesorjev podpira izvajanje večjega števila procesov naenkrat. Računalnik, na katerem je bilo izdelano testiranje, omogoča izvajanje do štirih procesov naenkrat, saj ima dve jedri in štiri niti.

V prihodnosti bi bilo zanimivo raziskovati uporabo kompaktnih priponskih dreves za reševanje problema, kot je iskanje v drsečem oknu. Ta problem se do sedaj rešuje s priponskimi drevesi. Zato je potrebno izdelati dinamično različico kompaktnih priponskih dreves, ki omogoča dodajanje in brisanje pripon iz drevesa. Druga zanimiva raziskava pa je izdelava kompaktnega priponskega drevesa, ki potrebuje konstanten čas za dostop do podnizov besede. To bi omogočalo, da se poizvedba  $seznamPojavov(T, P)$  izvede v času  $O(m + occ)$ , to je čas, ki ga ta poizvedba potrebuje v priponskem drevesu.

## 8 LITERATURA IN VIRI

- [1] E. UKKONEN, On-line construction of suffix trees. *Algorithmica* 14 (1995) 249–260. (*Citirano na strani 23.*)
- [2] K. SADAKANE, Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems* 41 (2007) 589–607. (*Citirano na straneh 46, 47, 48, 49, 51 in 52.*)
- [3] N. VÄLIMÄKI, W. GERLACH, K. DIXIT in V. MÄKINEN, Engineering a Compressed Suffix Tree Implementation. V *6th International Workshop on Experimental and Efficient Algorithms*, 2007, 217–228. (*Citirano na straneh 48, 56, 60 in 72.*)
- [4] E. M. MCCREIGHT, A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery* 23 (1976) 262–272. (*Citirano na strani 23.*)
- [5] P. WEINER, Linear pattern matching algorithms. V *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, 1973, 1–11. (*Citirano na straneh 1 in 56.*)
- [6] G. NAVARRO, *Compact Data Structures: A Practical Approach*, Cambridge University Press, 2016. (*Citirano na straneh 1, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 16, 18, 49, 50, 51, 54, 55, 56, 57 in 58.*)
- [7] D. E. KNUTH, J. H. MORRIS in V. R. PRATT, Fast Pattern Matching in Strings. *SIAM Journal on Computing* 6 (1977) 323–350. (*Citirano na strani 19.*)
- [8] D. GUSFIELD, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997. (*Citirano na strani 19.*)
- [9] *Pizza&Chili Corpus – Compressed Indexes and their Testbeds*, <https://pizzachili.dcc.uchile.cl/>. (Datum ogleda: 8. 10. 2024.) (*Citirano na straneh 60 in 61.*)
- [10] I. CANKAR, *Na Klancu*, Genija, 2012. (*Citirano na strani 61.*)
- [11] S. GOG, T. BELLER, A. MOFFAT in M. PETRI, From Theory to Practice: Plug and Play with Succinct Data Structures. V *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, 326–337. (*Citirano na strani 63.*)

- [12] K. GANESH, *ganesh-k13/suffix-tree: C++ implementation of Suffix Tree (Ukkonens)*, <https://github.com/ganesh-k13/suffix-tree>. (Datum ogleda: 8. 10. 2024.) (*Citirano na strani 63.*)
- [13] E. MILLER, *Genome — Knowledge Hub*, <https://www.genomicseducation.hee.nhs.uk/genotes/knowledge-hub/genome>. (Datum ogleda: 30. 10. 2024.) (*Citirano na strani 46.*)
- [14] T. KASAI, G. LEE, H. ARIMURA, A. SETSUO in K. PARK, Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching* 2089 (2001) 181-192. (*Citirano na straneh 39, 40, 41 in 58.*)
- [15] J. I. MUNRO in V. RAMAN, Succinct representation of balanced parentheses, static trees and planar graphs. *Proceedings 38th Annual Symposium on Foundations of Computer Science* (1997) 118-126. (*Citirano na strani 48.*)
- [16] R. GROSSI in J. S. VITTER, Compressed suffix arrays and suffix trees with applications to text indexing and string matching . *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing* (2000) 397–406. (*Citirano na straneh 49 in 52.*)
- [17] R. GROSSI, A. GUPTA in J. S. VITTER, High-Order Entropy-Compressed Text Indexes . *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2003) 841–850. (*Citirano na straneh 49 in 52.*)
- [18] L. M S. RUSSO, G. NAVARRO in A. L. OLIVEIRA, Fully-Compressed Suffix Trees. *LATIN 2008: Theoretical Informatics* (2008) 362–373. (*Citirano na strani 53.*)
- [19] D. HAREL in R. E. TARJAN, Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing* 13 (1984) 338-355. (*Ni citirano.*)
- [20] KOUTE, *koute/bytehound: A memory profiler for Linux.*, <https://github.com/koute/bytehound>. (Datum ogleda: 30. 10. 2024.) (*Citirano na strani 64.*)
- [21] A. JACOBS, The Pathologies of Big Data: Scale up your datasets enough and all your apps will come undone. What are the typical problems and where do the bottlenecks generally surface?. *Queue* 7 (2009) 10–19. (*Citirano na strani 70.*)
- [22] M. L. FREDMAN in D. E. WILLARD, BLASTING through the information theoretic barrier with FUSION TREES. V *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, 1990, 1-7. (*Citirano na strani 4.*)

- [23] P. MORIN, *Open Data Structures, An Introduction*, Athabasca University Press, 2013. (*Citirano na strani 4.*)
- [24] U. MANBER in G. MYERS, Suffix arrays: a new method for on-line string searches. V *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, 1990, 319-327. (*Citirano na straneh 37 in 39.*)
- [25] P. KO in S. ALURU, Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms* 3 (2005) 143-156. (*Citirano na straneh 39, 40 in 63.*)
- [26] M. I. ABOUELHODA, S. KURTZ in E. OHLEBUSCH, Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* 2 (2004) 53-86. (*Citirano na straneh 40, 41 in 43.*)
- [27] D. KNUTH, *Art of Computer Programming, The: Combinatorial Algorithms, Volume 4A*, Addison-Wesley Professional, 2011. (*Citirano na strani 6.*)
- [28] J. FISCHER in V. HEUN, A new succinct representation of RMQ-information and improvements in the enhanced suffix array. V *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, 2007, 459-470. (*Citirano na strani 40.*)
- [29] A. BRODNIK, *Searching in Constant Time and Minimum Space*, Ph.D. dissertation, University of Waterloo, 1995. (*Citirano na strani 6.*)
- [30] I. GREBNOV, *GitHub - IlyaGrebnev/libsa: The libsa library provides fast linear-time construction of suffix array (SA), generalized suffix array (GSA), longest common prefix (LCP) array, permuted LCP (PLCP) array, Burrows-Wheeler transform (BWT) and inverse BWT based on the induced sorting algorithm with optional OpenMP support for multi-core parallel construction.*, <https://github.com/IlyaGrebnev/libsa>. (Datum ogleda: 30. 6. 2025.) (*Citirano na strani 63.*)
- [31] N. TIMOSHEVSKAYA in W. FENG, SAIS-OPT: On the characterization and optimization of the SA-IS algorithm for suffix array construction. V *IEEE 4th International Conference on Computational Advances in Bio and Medical Sciences, ICCABS*, 2014, 1-6. (*Citirano na strani 63.*)