

UNIVERZA NA PRIMORSKEM  
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN  
INFORMACIJSKE TEHNOLOGIJE

Magistrsko delo  
**Kompaktna priponska drevesa**  
(Compressed 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, april 2025

## Ključna dokumentacijska informacija

Ime in PRIIMEK: Jani SUBAN

Naslov magistrskega dela: Kompaktna priponska drevesa

Kraj: Koper

Leto: 2025

Število listov: 73

Število slik: 17

Število tabel: 6

Število referenc: 21

Mentor: prof. dr. Andrej Brodnik

Somentor: izr. prof. dr. Rok Požar

UDK:

Ključne besede:

POPRAVI: Math. Subj. Class. (2010):

Izvleček:

POPRAVI:: Izvleček predstavlja kratek, a jedrnat prikaz vsebine naloge. V največ 250 besedah nakažemo problem, metode, rezultate, ključne ugotovitve in njihov pomen.

## Key document information

Name and SURNAME: Jani SUBAN

Title of the thesis: Compressed Suffix Tree

Place: Koper

Year: 2025

Number of pages: 73

Number of figures: 17

Number of tables: 6

Number of references: 21

Mentor: prof. dr. Andrej Brodnik

Co-Mentor: izr. prof. dr. Rok Požar

UDC:

Keywords:

Math. Subj. Class. (2010):

Abstract:

Izvleček predstavlja kratek, a jedrnat prikaz vsebine naloge. V največ 250 besedah nakažemo problem, metode, rezultate, ključne ugotovitve in njihov pomen.

# Kazalo vsebine

<b>1</b>	<b>UVOD</b>	<b>1</b>
1.1	STRUKTURA . . . . .	1
<b>2</b>	<b>NOTACIJA IN OSNOVE</b>	<b>3</b>
2.1	BITNO POLJE . . . . .	3
2.2	ENIŠKI ZAPIS . . . . .	10
2.3	KOMPAKтна PREDSTAVITEV DREVES . . . . .	10
2.3.1	Zaporedje eniških zapisov stopenj vozlišč po plasteh . . . . .	11
2.3.2	Zaporedje eniških zapisov stopenj vozlišč v globino . . . . .	13
2.3.3	Uravnoteženi oklepaji . . . . .	15
<b>3</b>	<b>PRIPONSKA DREVESA</b>	<b>17</b>
3.1	IZGRADNJA . . . . .	18
3.2	POIZVEDBE . . . . .	28
<b>4</b>	<b>PRIPONSKO POLJE</b>	<b>32</b>
4.1	SKUPNA NAJDALJŠA PREDPONA . . . . .	33
4.1.1	Simulacija priponskega drevesa . . . . .	34
4.2	IZGRADNJA . . . . .	39
4.3	POIZVEDBE . . . . .	40
<b>5</b>	<b>KOMPAKтна PREDSTAVITEV PRIPONSKEGA DREVESA</b>	<b>43</b>
5.1	IZGRADNJA . . . . .	50
5.2	POIZVEDBE . . . . .	53
<b>6</b>	<b>OPERACIJE NAD PRIPOSKIMI DREVSESI</b>	<b>56</b>
6.1	TEORETIČNA ANALIZA . . . . .	56
6.2	OPIS METODE EMPIRIČNE PRIMERJAVE . . . . .	64
6.2.1	Pred obdelava besedil . . . . .	67
6.2.2	Iskanje vzorcev . . . . .	67
6.3	REZULTATI EMPIRIČNE PRIMERJAVE . . . . .	68
<b>7</b>	<b>ZAKLJUČEK</b>	<b>82</b>

## 8 LITERATURA IN VIRI

84

# Kazalo preglednic

1	Implementacija operacij drevesa v LOUDS. . . . .	13
2	Implementacija operacij drevesa z DFUDS . . . . .	14
3	Implementacija operacij drevesa z BP . . . . .	16
4	Časovna zahtevnost operacij priponskega drevesa, izgradnje priponskega drevesa in iskanja v priponskem drevesu ter prostorska zahtevnost priponskega drevesa. . . . .	57
5	Primerjava prostorske zahtevnosti ter časovne zahtevnosti različnih implementacij kompaktne priponskega polja. . . . .	57
6	Primerjava besedil, ki bodo uporabljena za primerjavo različnih implementacij priponskih dreves . . . . .	66

# Kazalo slik in grafikonov

1	Primer predstavitve drevesa z metodo LOUDS za priponsko drevo besede »KOKOŠ\$«.	12
2	Primer predstavitve drevesa z metodo DFUDS za priponsko drevo besede »KOKOŠ\$«.	13
3	Primer predstavitve drevesa z metodo BP za priponsko drevo besede »KOKOŠ\$«.	15
4	Primer priponskega drevesa nad besedo »KOKOŠ\$«.	18
5	Primer izgradnje priponskega drevesa z uporabo Naivne metode za besedo »KOKOŠ\$«.	19
6	Primer izgradnje priponskega drevesa z uporabo Izboljšane naivne metode za besedo »KOKOŠ\$«.	21
7	Primer izgradnje priponskega drevesa z uporabo McCreightvega algoritma za besedo »KOKOŠ\$«.	22
8	Načini dodajanja novih znakov v priponsko drevo. Na sliki kvadratki predstavljajo liste drevesa, krogci pa predstavljajo notranja vozlišča drevesa.	25
9	Primer izgradnje priponskega drevesa z uporabo Ukkonenovaga algoritma za besedo »KOKOŠ\$«.	26
10	Primer priponskega polja nad besedo »KOKOŠ\$«.	32
11	Primer $L$ -LCP in $R$ -LCP polji za priponskega polja nad besedo »KOKOŠ\$«.	34
12	Primer intervalnega drevesa nad priponskim poljem in LCP poljem besede »KOKOŠ\$« ter njegova predstavitev s tabelo. Na sliki je dodano tudi priponsko drevo besede »KOKOŠ\$«.	36
13	Primer priponskega drevesa (levo) in kompaktnega priponskega drevesa (desno) za besedo »KOKOŠ\$«.	49
14	Zasedenost spomina testiranjem različnih implementacij priponskega drevesa skozi celotno izvajanje testa.	66
15	Posnetek zaslona upravljalnika opravil Htop med izgradnjo priponskega drevesa za besedilo dolžine 2048000 znakov.	69

16	Graf velikosti priponskega drevesa izgrajenih iz besedil različne velikosti. Vhodno besedilo je DNK sekvenca. . . . .	71
17	Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je DNK sekvenca. . . . .	72
18	Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je DNK sekvenca. . .	74
19	Graf velikosti priponskega drevesa izgrajenih iz besedil različnih velikosti. Vhodno besedilo je roman Na klancu. . . . .	76
20	Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je roman Na klancu. . . . .	77
21	Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je roman Na klancu.	78



# 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 querry</i>
rMq	angl. <i>range maximum querry</i>
LOUDS	angl. <i>Level Order Unary Degree Sequence</i> (slo. Zaporedje eniških zapisov stopenj vozlišč po plasteh)
DFUDS	angl. <i>Depth Frst 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 Preffix</i> (slo. Najdaljša skupna predpona)
BWT	angl. <i>Burrows–Wheeler Transform</i> (slo. Burrows–Wheelerjeva preslikava)
CST	angl. <i>Compressed Suffix Tree</i> (slo. Kompaktno priponsko drevo)
CSA	angl. <i>Compressed Suffix Array</i> (slo. Kompaktno priponsko polje)
CSV	angl. <i>Comma-separated values</i>
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

Tu se zahvalimo sodelujočim pri zaključni nalogi, osebam ali ustanovam, ki so nam pri delu pomagale ali so delo omogočile. Zahvalimo se lahko tudi mentorju in morebitnemu somentorju.

# 1 UVOD

Pogosti problem pri obdelavi daljših besedah je iskanje vzorcev v njih. V procesiranju naravnih jezikov se ta problem pretvori na iskanje vseh ponovitev besede v besedilu. Medtem ko se v bioinformatiki ta problem pretvori na preverjanje prisotnost specifičnih genov ali drugih DNK sekvenc v genomu.

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 prehoda 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 velikostjo besede, torej pri velikih besedah preraste velikost delovnega spomina, 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 kot tudi empirična primerjava različnih implementacij priponskih dreves.

## 1.1 STRUKTURA

Magistrska naloga je razdeljena na štiri dele. V drugem poglavju bodo predstavljene notacije uporabljen v magistrski nalogi. Predstavljen bodo tudi osnovne podatkovne strukture in operacije nad njimi, ki bodo uporabljen 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 sproti (angl. *on-line*) zgradi priponsko drevo v času  $O(n)$ . Predstavljeno bo tudi priponsko polje (angl. *Suffix Array* oziroma SA), ki se ga lahko

uporablja namesto priponskega drevesa za indeksirat vhodno besedo.

V četrtem 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 bodo tudi predstavljen algoritem za izgradnjo kompaktnega priponskega drevesa, ki skozi celotno izgradnjo ohranja kompaktnost podatkovne strukture.

V petem poglavju bodo predstavljene vse operacije nad priponskimi drevesi. Prikazana bo teoretična primerjava različnih implementacij. Pri tem bo izdelana tudi empirična primerjava implementaciji priponskih dreves, ki bo merila prostorsko in časovno zahtevnost.

## 2 NOTACIJA IN OSNOVE

Skozi magistrsko nalogo  $T$  predstavlja vhodno besedo dolžine  $n$  znakov, iz katere bomo zgradili priponsko drevo ali kompaktno priponsko drevo. Na podoben način  $P$  predstavlja iskani vzorec dolžine  $m$  znakov v  $T$ . Vhodna beseda  $T$  in vzorec  $P$  sta sestavljena iz znakov abecede  $\Sigma$ .

Niz znakov, ki je del besede in se začne na  $i$ -tem znaku besede, imenujemo podniz in je 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  $\alpha = T[i, i + |\alpha|]$ .

Poljubni nizi oziroma podnizi so označeni z grškimi črkami  $\alpha$ ,  $\beta$  in  $\gamma$  ter pri tem je povedano, ali gre za podniz ali niz. Podniz besede  $T$ , ki se začne na  $i$ -tem mestu in se konča na  $j$ -tem mestu, pa je označen kot  $T[i, j]$ . Dva poljubna niza se lahko združita v novi daljši niz s stikom, ki je definiran na sledeči način:

**Definicija 2.2.** Stik nizov  $\alpha$  in  $\beta$  je niz  $\gamma = \alpha \cdot \beta$ , 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  $l$ .

V nadaljevanju poglavja bodo predstavljena še predstavitev in podatkovne strukture ter operacije nad njimi, ki so osnova za kompaktno predstavitev drugih podatkovnih struktur (na primer kompaktna predstavitev dreves), ter kompaktne predstavitev dreves, pri čemer bo ena od predstavitev uporabljena v podatkovni strukturi kompaktno priponsko drevo. Predstavlja bo tudi eniški zapis, ki bo uporabljen v kompaktni predstavitvi priponskih dreves.

### 2.1 BITNO POLJE

Bitno polje (angl. *bit vector*)  $B$  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 kompaktnih predstavitev

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, ki so predstavljeni z bitnimi polji. Predstavljene operacije so:  $\text{rang}(B, i)$  (angl. *rank*),  $\text{izbira}(B, i)$  (angl. *select*),  $\text{predhodnik}(B, y)$ ,  $\text{naslednik}(B, y)$  in dodatne operacije nad intervali (angl. *range query*) bitnega polja. Vse predstavljene operacije želimo opraviti čim bolj učinkovito ter pri tem uporabiti čim manj dodatnega prostora.

**Model računanja, shramba in dostop:** Preden se lahko predstavi način shranjevanja bitnega polja, je potrebno predstaviti uporabljen model računanja. Splošen model računanja bo uporabljen, saj se različne arhitekture mikro procesorjev razlikujejo med seboj in bi bilo potrebno narediti analizo za vsako arhitekturo posebj. Zato bo izbran model računanja Računalnik z naključnim dostopom (angl. *random-access machine* oziroma RAM). Bolj natančno bo uporabljen besedni RAM (angl. *word RAM*).

Osnovna različica RAM modela podpira, da so vse aritmetične operacije nad celimi števili in logične operacijami nad biti izvedene v konstantnem času. Poleg tega RAM omogoča dostopa do spomina v konstantnem času ter linearni čas za dodelitev zaporednega spomina. Vse te operacije so storjene v enakem času tudi v besednem RAM modelu. Edina razlika med modeloma je, da RAM predpostavlja neskončni spomin za razliko od besednega RAM, ki pa omeji velikost spomina na  $U$  naslovov in posledično je velikost ene besede na  $w = \log_2 U^1$  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  $w$  biti [6].

Intuitivni način shranjevanja bitnega polja  $B$  dolžine  $b$  v spominu bi bil tak, da bi se vsaka celica polja shranila v spominu na lastnem naslovu. Na ta način bo operacija  $\text{dostop}(B, i) = B[i]$  potrebovala konstanten čas, da se izvede. Pri tem pa bo bitno polje potrebovalo  $O(b)$  bitov oziroma  $wb$  bitov. Na ta način imam vsaka celica bitnega polja  $w - 1$  odvečnih bitov.

Zato se pojavi vprašanje, ali je mogoče izogniti teh  $b(w - 1)$  odvečnih bitov. Če 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 velikosti »celega števila«  $w$ ). Torej bi se lahko bitno polje  $B$  predstavilo kot polje »celih števil«  $W[1, \lceil \frac{b}{w} \rceil]$ . 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. Le ta je shranjen v celici  $W[\lceil \frac{i}{w} \rceil] = w_i$ . Naivna metoda dostopa do bita v »celem številu« bi bila z bitnim premikom (angl. *bit shift*) v desno,

<sup>1</sup>v nadaljevanju bo  $\log_2 n$  zapisan kot  $\log n$

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 enakovren celoštevilskemu deljenju števila z dva, se lahko nadomesti premike v desno s celoštevilskemu deljenjem števila  $w_i$  s številom  $2^{w-r}$ . Torej dostop do  $i$ -tega bita v bitnem polju  $B$  izračunamo kot:

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

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

**Rang:** Prva operacija, ki se jo lahko definira 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 se lahko naivno implemetira s štetjem bitov z vrednostjo 1 v bitnem polju, za kar je potrebno  $O(b)$  časa. Operacijo rang se lahko pospeši na konstantni čas, pri tem pa se potrebuje 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, ki omogoča konstantni čas operacije  $\text{rang}(B, i)$ , 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 pomožnosti ni večji od  $o(b)$  bitov. Predlagana rešitev potrebuje  $b \log r_1 = O(bw)$  dodatnih bitov, pri čemer  $r_1$  predstavlja 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  $0 \leq i \leq \lfloor \frac{b}{s} \rfloor$ . Na ta način je operacija rang implementiran kot

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

kjer je **popcount** funkcija, ki prešteje število bitov z vrednostjo 1 na danem intervalu. Zaradi uporabnosti te funkcije je le ta že implementirana v različnih modernih arhitekturah procesorjev in je zato možno uporabiti strojno operacijo. Sicer pa je mogoče

implementirati funkcijo **popcount** brez uporabe strojne operacije [27]. 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** pognati največ  $k$  krat, kar naredi čas operacije rang  $O(k)$  [6].

Podobno kot polje  $R$  se definira polje  $R'$ . Polje  $R'$  hrani na  $i$ -tem mestu število bitov z vrednostjo 1 po vedru  $R[\lfloor \frac{i}{k} \rfloor]$  ali  $R'[i] = rang_1(B, iw) - R[\lfloor \frac{i}{k} \rfloor]$ . Na ta način se zniža število klicev funkcije **popcount** na 1 klic. Ker so v  $R'$  shranjena zgolj števila med 0 in  $s - w$  (vsako vedro v  $R$  ima  $w$  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 polji  $R$  in  $R'$ . In sicer je implementirana na sledeči način:

$$rang_1(B, i) = R \left[ \left\lfloor \frac{i}{kw} \right\rfloor \right] + R' \left[ \left\lfloor \frac{i}{w} \right\rfloor \right] + \text{popcount} \left( B \left[ \left\lfloor \frac{i}{w} \right\rfloor, \left\lfloor \frac{i}{w} \right\rfloor + (i \bmod w) \right] \right).$$

Ta implementacija potrebuje konstanten čas za izračunati  $rang(B, i)$ . Dostop do polji  $R$  in  $R'$  je storjen v konstantnem času, kot tudi izračun funkcije **popcount**, ki je klicana samo enkrat [6].

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

Operacijo izbira si lahko predstavimo kot inverzno operacijo operacije rang, saj velja zveza  $j = rang_v(B, izbira_v(B, j))$ . Pri tem pa ne obstaja direktna povezava med operacijo  $izbira_1(B, i)$  in  $izbira_0(B, i)$ . To pomeni, da rešitev s pomožno podatkovno strukturo za  $izbira_1(B, i)$  ne more biti uporabljena pri iskanju rešitve za  $izbira_0(B, i)$ . V nadaljevanju bo opisan časovno učinkovit postopek iskanja  $izbire_1(B, i)$ , saj se lahko  $izbira_0(B, i)$  implementira na podoben način [6].

Ko operacija izbira ni ključna pri reševanju problemov, se lahko uporabi binarno iskanje v poljih  $R$  in  $R'$ , za kar se potrebuje  $O(\log b)$  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 operacij 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  $S[i] = izbira_1(B, i * s + 1)$ , pri čemer  $S[\lceil \frac{r_1}{s} \rceil] = n + 1$ . 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 kot  $s \log^2 n$  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  $V[i] = 0$ . Bitno polje  $V$  potrebuje tudi dodatno podatkovno strukturo za rang. Za velika vedra se izračuna vseh  $s$  vrednosti izbire, pri čemer so shranjeni



v polju  $I$  položaji bitov z vrednostjo ena znotraj vedra. Za mala vedra pa se sproti naračuna izbira znotraj vedra. Operacija izbira se izračuna na sledeči način:

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

$$b + 1, \quad m < j$$

kjer je  $x = ((j - 1) \bmod s) + 1$  in  $k$  je položaj  $x$ -tega bita z vrednostjo 1 na intervalu  $B \left[ S \left[ \left\lceil \frac{j}{s} - 1 \right\rceil \right], S \left[ \left\lceil \frac{j}{s} \right\rceil - 1 \right] \right]$ .

Polje  $I$  mora shraniti  $s \lceil \log n \rceil$  bitov za vsako veliko vedro, katerih 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 ter dodatne bite za izvajanje operacije rang v konstantnem času, torej tudi  $V$  potrebuje  $o(b)$  bitov [6].

Operacija se lahko izvede v konstantnem času. To je doseženo z dodatnim deljenjem majhnih veder, na podoben način, kot je bilo to storjeno nad celotnim bitnim poljem  $B$ . Vsako majhno vedro se dodatno razdeli na  $s' = (\log \log b)^2$  mini veder, pri čemer mini vedro je veliko, ko je večje od  $s'(\log \log b)^2$ . Vsako majhno mini vedro potrebuje  $s' \log(s(\log b)^2) = O((\log \log b)^3) = o(b)$  bitov ter vsako veliko mini vedro potrebuje isto prostora, pri čemer pa jih je največ  $O \left( \frac{b}{(\log \log b)^4} \right)$ , torej vsa velika mini vedra skupaj potrebujejo  $O \left( \frac{b}{\log \log b} \right) = o(b)$  bitov [6].

Vse dodatne podatkovne, ki so strukture 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 ves potreben spomin že predhodno dodeljen.

**Predhodnik/Naslednik:** S pomočjo operaciji rang in izbira se lahko implementira 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  $B[i] = v$ . Operacija je implementirana kot

$$predhodnik_v(B, y) = izbira_v(B, rang_v(B, y)),$$

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

$$naslednik_v(B, y) = izbira_v(B, rang_v(B, y - 1) + 1),$$

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

Operaciji se uporablja 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  $B$ ).

**Operacije na intervalih:** Dosedaj 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 največje ali najmanjše število v intervalu. Na bitnih poljih pa se ta problem pretvori na razliko med številom bitov z vrednostjo 0 ter številom bitov z vrednostjo 1 do  $i$ -tega bita. To razmerje imenujemo *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 višek na intervalu med indeksom  $i$  in  $j$ , in sicer kot:

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

Prva operacija na intervalih 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  $j$  [6].

Vse predstavljene operacije na intervalih (razen operacije *višek*, ki potrebuje konstanten čas) 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. Podatkovna struktura razdeli bitno polje  $B$  na  $\frac{b}{a}$  vedr velikosti  $a$  elementov ter se za vsako vedro ustvari list rmM-drevesa. 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 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  $\text{rmq}(B, i, j)$  na intervalu bitnega polja  $B[i, j]$ . Operacija  $\text{rmq}(B, i, j)$  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 naračuna relativni višek  $d$  ter 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 s sprehodom 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 straša, 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 prejšni pogoj, pa nadaljujemo s sprehodom navzdol iz našega desnega brata  $v \leftarrow (v + 1)/2$ . Tokrat pa preverjamo, ali je levi otrok vsebovan v intervalu ter če vozlišče  $v$  omogoča zmanjševanje najmanjšega viška. Če vozlišče  $v$  ne omogoča zmanjševanja, lahko sprehod končamo in vemo, da najmanjši višek je  $m$ , sicer pa nadaljujemo s sprehodom. Če levi otrok ni vsebovan, nadaljujemo s sprehodom v levem otroku, sicer pa preverimo, če velja  $m > d + rmM[2v].m$  ter v primeru, da velja popravimo vrednost  $m$ . Nato popravimo vrednost  $d \leftarrow d + rmM[2v].e$  ter nadaljujemo z iskanjem v desnem otroku  $v \leftarrow 2v + 1$ . Sprehod nadaljujemo dokler ne dosežemo listov oziroma  $v > \lceil n/a \rceil$ . 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  $R[v].m$ , 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 (vsak 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].

## 2.2 ENIŠKI ZAPIS

Preden se lahko bitno polje uporabi za kompaktno predstavitev dreves, je potrebno predstaviti še eniški zapis. Eniški zapis je uporabljen v dveh kompaktnih predstavitev dreves. Kot je iz samega imena zapisa razvidno, bodo števila predstavljena v eniškem sistemu. Torej bo nenegativno celo število  $s$  predstavljeno s  $s$ -timi enicami ter števila so med seboj ločena z ničlo. Na primer, seznam števil  $[1, 3]$  bi bil v eniškem zapisu predstavljen kot 101110.

Razlog za uporabo eniškega zapisa števil namesto dvojiškega zapisa števil v kompaktni predstavitvi dreves je lažja pretvorba operacij nad drevesi na operacije nad bitimi polji, saj imajo na ta način enice in ničle vsaka svoj pomen. Ker je zapis uporabljen za predstavitev stopenj vozlišč, potem predstavlja zaporedje  $1^*0$  eno vozlišče. Vsaka enica v vozlišču predstavlja otroka vozišča, ničla pa predstavlja, da je bilo vozlišče že obiskano v zapisu.

## 2.3 KOMPAKтна PREDSTAVITEV DREVES

Z uporabo bitnih polj je mogoče predstaviti mnogo podatkovnih struktur. Ker pa se magistrska naloga osredotočila na priponska drevesa, bo zato predstavljena uporaba bitnih polj za kompaktno predstavitev topologije dreves. Običajno je podatkovna struktura drevo sestavljeno iz vozlišč ter kazalcev (angl. *pointers*). V vsakem vozlišču je shranjena vrednost, ki jo predstavlja vozlišče, ter kazalci 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 kazalec na vozlišče ter predstaviti, da je vozlišče bilo že obiskano, torej 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, ki so potrebne za pravilno delovanje dreves.

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

1.  $koren()$  vrne koren drevesa,
2.  $jeList(v)$  vrne *true*, če je vozlišče 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.  $stars(v)$  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  $w$ .

S pomočjo teh operacij se je mogoče sprehoditi po drevesu in je možno implementirati ostale operacije dreves, na primer *vstavi*, *izbriši*, *najmanjši\_element* (v kopici) in ostale.

### 2.3.1 Zaporedje eniških zapisov stopenj vozlišč po plasteh

Prvi način zapisa topologije drevesa je zaporedje eniških zapisov stopenj vozlišč po plasteh (LOUDS). Drevo 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 1. 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.

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. Vrednost  $x$  predstavlja indeks vozlišča in se izračuna kot  $x = izbira_0(B, v) + 1$  [6].



Slika 1: Primer predstavitve drevesa z metodo LOUDS za priponsko drevo besede »KOKOŠ\$«.

Vse operacije razen operacije *globina* in *lca* so izvršene v konstantnem času z uporabo dodatne podatkovne strukture za rang in izbiro. 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 uporabe operacije  $stars\check{s}(v)$ , dokler vozlišče  $v$  ni *koren* drevesa. Pri tem pa velja, da  $globina(u) \geq globina(v)$ , ko je  $u > v$ . S pomočjo tega dejstva se lahko implementira operacija *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      $v \leftarrow stars\check{s}(B, v)$ 
4   sicer
5      $w \leftarrow stars\check{s}(B, w)$ 
6 vrni  $v$ 
```

---

Tabela 1: Implementacija operacij drevesa v LOUDS.

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

### 2.3.2 Zaporedje eniških zapisov stopenj vozlišč v globino

Naslednja predstavitev topologije drevesa je zaporedje eniških zapisov stopenj vozlišč v globino (DFUDS). Predstavitev omogoča preprostejše implementacije operacij ter posledično manjše dodatne podatkovne strukture. Vseeno pa omogoča hitrejšje implementacije nekaterih operacij v primerjavi z uporabo predstavitve LOUDS. Primer zapisa topologije drevesa z DFUDS je prikazan na Sliki 2. Vozlišča so v zaporedju ločena s sivimi črtami.



Slika 2: Primer predstavitve drevesa z metodo DFUDS za priponsko drevo besede »KOKOŠ\$«.

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.

Tabela 2: Implementacija operacij drevesa z DFUDS

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

V Tabeli 2 so predstavljene implementacije operaciji nad drevesom, implementirane z DFUDS. Indeks  $x = izbira_0(B, v) + 1$  je uporabljen za shranjevanje dodatnih informacij o vozlišču, saj je vozlišče  $v$  predstavljeno s položajem vozlišča v bitnem polju  $B$ . 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  $izbira_1$ ,  $izbira_0$ , *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  $rang_{00}$  in  $izbira_{00}$  v konstantnem času, ki omogočata iskanje in indeksiranje listov v drevesu. Podatkovni strukturi sta implementirani na podoben način kot podatkovni strukturi za  $rang_0$  in  $izbira_0$ , pri tem pa  $rang_{00}$  in  $izbira_{00}$  temeljijo na številu ponovitev oziroma položaju  $i$ -te ponovitve dveh zaporednih bitov z vrednostjo 0.



### 2.3.3 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 3. 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].



6 1 3 2 4 5  
001001011001011011

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

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  $B$ . Operacijo  $odpri(B, i)$  se lahko definira tudi tako, da vrne zadnji  $j < i$ , pri čemer  $višek(B, j) = višek(B, i) + 1$ . Podobno operacija  $zapri(B, i)$  vrne položaj ')', ki zapre '(' na  $i$ -tem mestu v  $B$ . Z uporabo operacije  $višek$  je operacija  $zapri$  definirana tako, da vrne prvi  $j > i$ , pri čemer je  $višek(B, j) = višek(B, i) - 1$ . Operacija  $oklepa(B, i)$  vrne položaj  $j$  od '(' v bitnem polju  $B$ , pri čemer velja, da  $j < i < zapri(B, j)$ . Z uporabo operacije  $višek$  operacija

$oklepa(B, i)$  vrne položaj največjega  $j < i$ , pri čemer je  $višek(B, j-1) = višek(B, i) - 1$  [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  $rang_1$  in  $izbira_1$  nista potrebni za pravilno delovanje drevesa v tej predstavitvi. Podatkovni strukturi potrebujeta vsaka  $o(n)$  dodatnih 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  $x = izbira_0(B, v)$  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  $\rangle() \langle$  (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 poizvedba  $rmq(B, i, j)$ , 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 na pomnilniku kot polje črk  $T[1..n]$ , ampak zakodira tudi njegovo strukturo. Zato se pogosto uporabi 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, ki potrebuje  $O(n + m)$  časa za preveriti, ali se vzorec nahaja v besedi. Za razliko od KMP algoritma je prednost uporabe priponskega drevesa, da lahko iščemo različne vzorcev v isti besedi  $T$ . Za najti več vzorcev v 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].

Priponsko drevo nad besedo, ki je niz nad abecedo  $\Sigma$ , definiramo na sledeči način [8]:

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

1. drevo ima natanko  $n$  listov oštevilčenih s števili med 1 in  $n$ ,
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. podniz, ki je pridobljen s stikom podnizov na poti od korena do lista  $i$ , predstavlja pripono  $T[i, n]$  za vsak  $i$ , kjer je  $1 \leq i \leq n$ .

Primer priponskega drevesa besede »KOKOŠ« je prikazan na Sliki 4. 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 4, znak »\$« ne bi bil potreben, ker že znak »Š« jasno določi vse pripone besede »KOKOŠ«. Da zagotovimo, da vse so vse pripone shranjene v listih, se besedi na konec pripne znak »\$«.



Slika 4: Primer priponskega drevesa nad besedo »KOKOŠ\$«.

Ker povezave v priponskih drevesih predstavljajo podnize v besedi  $T$ , se lahko nad priponskimi drevesi definira tudi črkovna globina vozlišča. Ta je definirana na sledeči način:

**Definicija 3.2.** Črkovna globina vozlišča  $Sd(v)$  je dolžina podniza, pridobljenega s stikom vseh podnizov na povezavah na poti od korena drevesa do vozlišča  $v$ .

Primer razlike med globino vozlišča in črkovno globino vozlišča se lahko vidi na Sliki 4: 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 IZGRADNJA

V tem podpoglavju bodo predstavljene različne metode izgradnje priponskih dreves. Metode bodo predstavljene v zaporedju od najbolj počasne, ki izgradi priponsko drevo v času  $O(n^3)$ , do najhitrejše, ki izgradi drevo v času  $O(n)$ .

**Naivna metoda:** Ta metoda v vsakem koraku s prehodom po drevesu podaljša vse pripone za en znak ter doda novo pripono. Metoda v  $i$ -tem koraku izgradnje v dosedaj izgrajeno drevo, ki je bilo izgrajeno za podniz  $T[1, i-1]$ , doda znak  $T[i]$  ter tako izgradi drevo za podniz  $T[1, i]$ . To je storjeno z dodajanjem pripone, ki predstavlja znak  $T[i]$ , in s podaljševanjem pripon, ki že obstajajo v drevesu. Ker listi vedno predstavljajo konec pripone, se niz na povezavi, ki vodi v list, podaljša za en znak.

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



Slika 5: Primer izgradnje priponskega drevesa z uporabo Naivne metode za besedo »KOKOŠ\$«.

eksplicitno predstavljene. Kljub temu pa to drevo vsebuje vso informacijo o pripadajočih priponah, pri čemer moramo posebej beležiti vse implicitno predstavljene pripone. Takšnemu drevesu rečemo implicitno priponsko drevo. Problem implicitno predstavljenih pripon v  $i$ -tem koraku se reši z beleženjem takih pripon. Če se niz na povezavi, ki predstavlja zadnji del implicitne pripone, nadaljuje po priponi z znakom  $T[i]$  se ne stori ničesar, saj je tudi podaljšana pripona implicitno predstavljena. Če pa se niz na povezavi, ki predstavlja zadnji del implicitne pripone, ne nadaljuje z znakom  $T[i]$ , pa se povezavo razdeli na dva dela in se ustvari vozlišče, na katerega kaže povezava z nizom, ki se ujema s predhodnjo pripono, ter ima dva otroka: vozlišče, na katerega kaže preostanek predhodnje povezave, ter list z znakom  $T[i]$ . V primeru, da vozlišče že obstaja, se le temu doda novega otroka, na katerega kaže povezava z znakom, ki se razlikuje. Postopek izgradnje priponskega drevesa za besedo »KOKOŠ\$« z naivno metodo izgradnje je prikazan na Sliki 5. Na Sliki 5 zgolj tretje (zadnje drevo v prvi vrstici) in četrto (prvo drevo v drugi vrstici) drevo sta implicitna priponska drevesa.

**Izrek 3.3.** 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 glo-

bina 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  $\sum_{j=1}^i j = \frac{i(i+1)}{2}$  vozlišč.

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

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

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

**Izboljšana naivna metoda:** Naivna metoda je preprosti način izgradnje priponskega drevesa, vendar se hitro opazijo načini za pospešitev izgradnje drevesa. Najprej opazimo, da metoda nepotrebno pregleduje celotno drevo. To se lahko reši z uporabo priponskih povezav (angl. *Suffix link*).

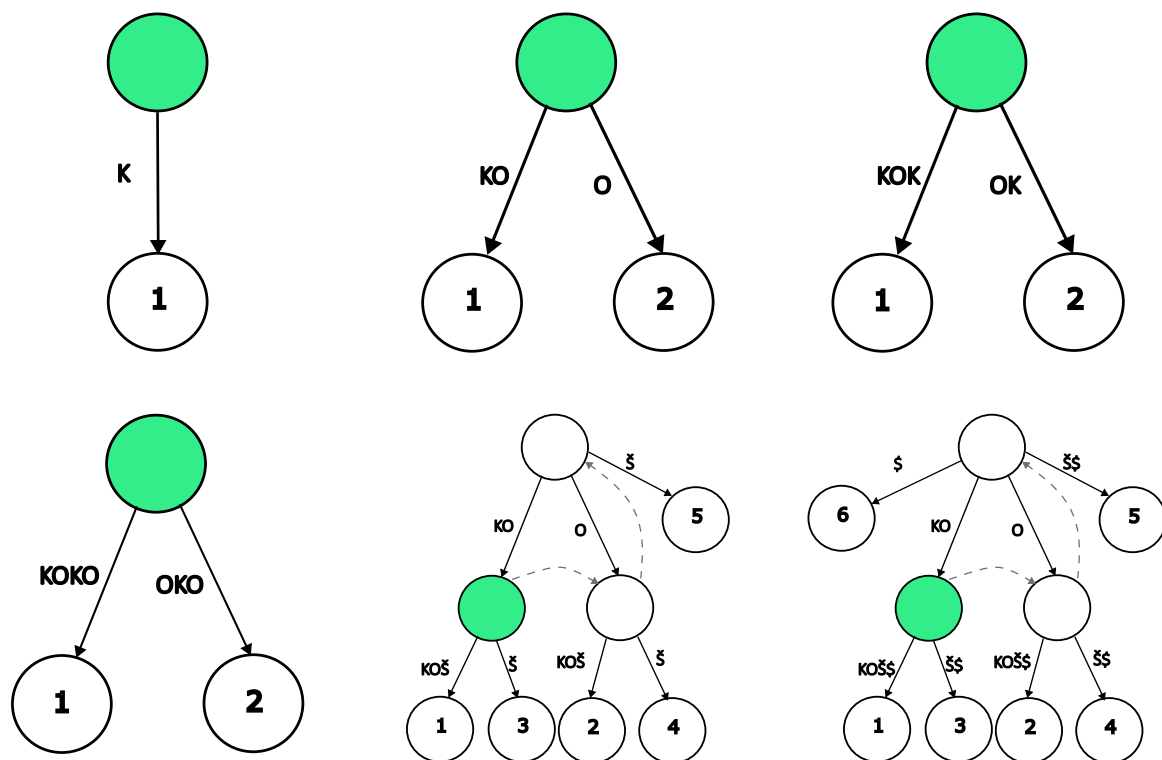
**Definicija 3.4.** Naj notranje vozlišče  $v$  predstavlja niz  $x \cdot \alpha$ , kjer je  $x$  znak in  $\alpha \in \Sigma^*$  je podniz niza  $T$ . Priponska povezava  $sl(v)$  je povezava na notranje vozlišče  $w$ , ki predstavlja niz  $\alpha$ .

Na Sliki 4 sta priponski povezavi prikazani s črtkano puščico. Uvedba priponskih povezav omogoča izogib nepotrebnim sprehodom po drevesu. Tako metoda podaljša vse liste zgolj s sprehodom po priponskih povezavah. Pri tem se opazi, da metoda lahko beleži notranje vozlišče s povezavo na list, ki predstavlja najdaljšo pripono v predhodnjem koraku izgradnje  $T[1, i-1]$ . Beleženje tega vozlišča, ki ga imenujemo začetna točka, odstrani nepotrebno iskanje začetka sprehoda po priponskih povezavah.

Metoda mora še vedno beležiti implicitne pripone (pripone, ki so predpone drugim priponam), ampak to beleženje ne vpliva na čas izgradnje priponskega drevesa. Metoda se v vsakem koraku sprehodi iz začetne točke po priponskih povezavah do korena. Postopek izgradnje priponskega drevesa za besedo »KOKOŠ« z izboljšano naivno metodo izgradnje je prikazan na Sliki 6. Na sliki je začetna točka označena z zeleno barvo. Ker je v  $i$ -tem koraku izgradnje največ  $i$  eksplicitno definiranih pripon, je tudi število priponskih povezav na poti iz začetne točke do korena po priponskih povezavah enako številu notranjih vozlišč  $O(i)$ . Torej je čas  $i$ -tega koraka  $O(i)$ . Potemtakem ta metoda potrebuje  $O(n^2)$  časa za izgradnjo celotnega drevesa. Iz tega sklepa sledi izrek.

**Izrek 3.5.** Izboljšana naivna metoda zgradi priponsko drevo nad besedo  $T$  v času  $O(n^2)$ .

Čeprav te izboljšave znižajo čas izgradnje priponskega drevesa na  $O(n^2)$ , le taka izboljšava ne omogoča izgradnje drevesa v času  $O(n)$ . Za izgradnjo priponskega drevesa



Slika 6: Primer izgradnje priponskega drevesa z uporabo Izboljšane naivne metode za besedo »KOKOŠ\$«.

v linearnem času obstajata dva algoritma: McCreightov algoritem [4] in Ukkonenov algoritem [1].

**McCreightov algoritem:** McCreightov algoritem ali Algoritem M je prvi algoritem s časovno zahtevnostjo  $O(n)$  za izgradnjo drevesa, pri tem pa je tudi prostorska zahtevnost algoritma  $O(n)$ . Algoritem deluje, tako da se v  $i$ -tem koraku v do takrat izgrajeno drevo doda pripona  $T[i, n]$ . Torej so pripone dodane v drevo v vrstnem redu od najdaljše do najkrajše pripone, za razliko od naivne metode, ki v  $i$ -tem koraku izgradnje priponskega drevesa doda  $i$ -ti znak v do takrat izgrajeno priponsko drevo, ki je lahko implicitno. Postopek izgradnje priponskega drevesa za besedo »KOKOŠ« z McCreightvim algoritmom je prikazan na Sliki 7.

V vsakem koraku je pripona razdeljena na dva dela, in sicer na glavo ter rep. Glava, označena kot  $glava_i$ , je najdaljša predpona  $i$ -te pripone, ki že obstaja v drevesu. Če take pripone ni v drevesu, je glava pripone prazna. Rep besedila, označen kot  $rep_i$ , pa je definiran kot preostanek pripone, ki ni del glave. Za razliko od glave, rep ne more biti prazen. Ker algoritem temelji na dejstvu, da nobena pripona ni predpona drugi priponi, se mora  $n$ -ti znak besede razlikovati od ostalih znakov v besedi in se ga označi



Slika 7: Primer izgradnje priponskega drevesa z uporabo McCreightvega algoritma za besedo »KOKOŠ\$«.

z znakom »\$«. Algoritem v vsakem koraku doda krajšo pripono, zato je v repu vedno prisoten vsaj »\$«, saj se bo vedno razlikoval od vsakega znaka, ki je na  $(n - i)$ -tem mestu poljubne že umeščene pripone [4].

Vsakič, ko se niz  $glava_i$  ne konča v vozlišču, Algoritem M ustvari novo vozlišče. Ker  $rep_i$  ni še vsebovan v drevesu, se doda novo list v drevo s povezavo, ki predstavlja niz  $rep_i$ . Repi so dodani v vsakem koraku v konstantnem času, torej algoritem potrebuje zgolj  $O(n)$  časa za dodajanje vseh repov.

Glavo  $i$ -te pripone je možno razdeliti na  $glava_i = \alpha\beta\gamma$ , pri čemer je  $\gamma$  lahko prazen podniz. Podniza  $\alpha$  in  $\beta$  sta definirana kot dela  $glava_{i-1} = x\alpha\beta$ , pri čemer je  $x$  znak  $T[i - 1]$ . Podniz  $\alpha$  je prazen niz natanko tedaj, ko je vozlišče, ki predstavlja najdaljšo predpono od  $x\alpha$ , koren. Če je  $\alpha$  prazen, potem je vozlišče  $a$ , ki je začetna točka iskanja glave, koren. Sicer je vozlišče  $a = sl(b)$ , kjer je  $b$  vozlišče, ki predstavlja najdaljšo predpono niza  $x\alpha$ .

Naslednji korak izgradnje priponskega drevesa je iskanje niza  $\beta$  v drevesu. To imenujemo *rescanning*. Po definiciji  $glava_i$  že obstaja v drevesu, torej obstaja tudi pot iz  $a$  v  $c$ , ki se začne z nizom  $\beta$ . Iskanje poteka tako, da algoritem primerja dolžino  $|\beta|$  z dolžino niza na povezavi iz  $a$ . Če je  $|\beta|$  krajša ali enaka od dolžine niza na povezavi



iz  $a$ , se iskanje prekine. V primeru, da je  $|\beta|$  strogo krajša, se ustvari novo vozlišče  $d$ . Če je  $|\beta|$  daljša od dolžine niza na povezavi iz  $a$ , se izbriše podniz, ki je predstavljen na povezavi, in otrok od  $a$  postane novo vozlišče  $a$ . Postopek se ponovi, dokler  $|\beta|$  ni krajša ali enaka od niza, ki ga predstavlja povezava iz  $a$ . V primeru, da ne obstaja priponska povezava iz vozlišča, ki predstavlja niz  $x\alpha\beta$ , v vozlišče  $d$ , se jo ustvari.

Iz vozlišča  $d$  se začne iskanje  $\gamma$ , če le ta ni prazen niz. Ta operacija je imenovana *scanning*. Za razliko od niza  $\beta$  dolžina niza  $\gamma$  ni znana vnaprej. Algoritem mora zato previti vsak znak, dokler ne najde znaka, ki se razlikuje. Ta znak predstavlja prvi znak v repu. V točki, kjer se znaka razlikujeta, je ustvarjeno novo vozlišče, če ta točka ni že vozlišče. Na to vozlišče se pripne  $rep_i$ .

**Izrek 3.6.** *McCreightov algoritem zgradi priponsko drevo nad besedo  $T$  v času  $O(n)$ .*

*Dokaz.* McCreightov algoritem v vsakem koraku naredi tri operacije: operacija vstavljanja repa v drevo, operacijo *scanning* in operacijo *rescanning*. Operaciji *scanning* in *rescanning* sta v  $i$ -tem koraku opravljeni nad nizom  $\beta_i\gamma_i rep_i$ .

Za vstaviti rep v drevo algoritem potrebuje  $O(1)$  časa. V drevo je potrebno vstaviti  $n$  pripon, torej je potreben čas za vstaviti vse repe v drevo  $T_{rep} = O(n)$ .

Operacija *scanning* potrebuje v vsakem koraku  $|glave_i| - |glave_{i-1}| + 1 = |\gamma_i|$  časa. Torej po vseh korakih operacija *scanning* potrebuje  $T_{scan} = \sum_{i=1}^n |glave_i| - |glave_{i-1}| + 1 = n + |glava_n| - |glava_0| = O(n)$  časa.

V  $i$ -tem koraku operacija *rescanning* obiše  $n_i$  vozlišč. Pri tem se opazi, da v naslednjem koraku ta vozlišča ne bodo obiskana. Iz tega sledi:  $|\beta_{i+1}\gamma_{i+1} rep_{i+1}| \leq |\beta_i\gamma_i rep_i| - n_i$ , pri čemer je  $|\beta_n\gamma_n rep_n| = |rep_n| = 1$ . Torej velja tudi:

$$|\beta_n\gamma_n rep_n| = |rep_n| \leq |\beta_1\gamma_1 rep_1| - \sum_{i=1}^n n_i,$$

$$1 \leq n - \sum_{i=1}^n n_i,$$

$$n \geq 1 + \sum_{i=1}^n n_i.$$

Iz tega sledi, da operacija *rescanning* skozi celotno izgradnjo obiše največ  $n$  vozlišč, zato potrebuje  $T_{rescan} = O(n)$  časa skozi celotno izgradnjo.

Torej je potrebni čas za izgradnjo priponskega drevesa vsota časov potrebnih za posamezno operacijo:

$$T_{izgradnja} = T_{rep} + T_{rescan} + T_{scan} = O(n) + O(n) + O(n) = O(n).$$

□

Čeprav McCreightov algoritem zgradi priponsko drevo v času in prostoru  $O(n)$ , algoritem predpostavi, da je beseda  $T$  vnaprej poznana. Pri tem se pojavi vprašanje, ali

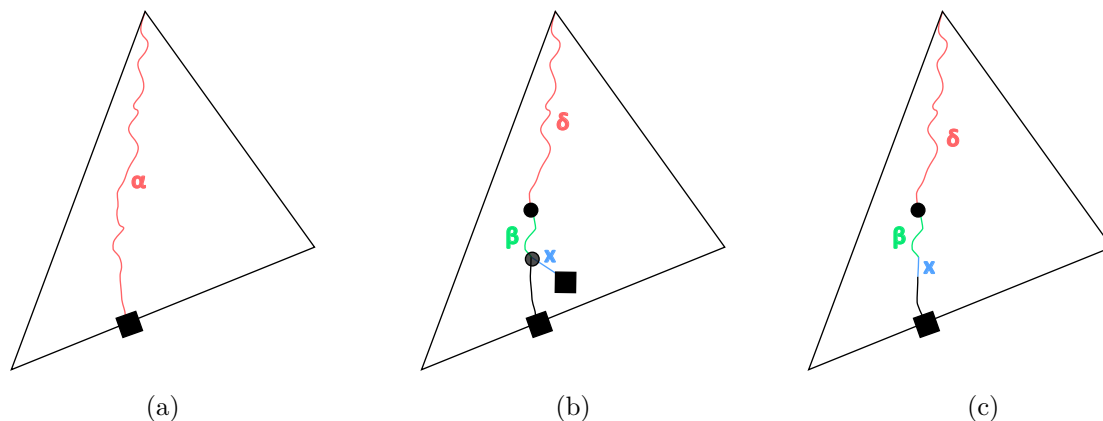
je mogoče izgraditi priponsko drevo, ne da bi vnaprej poznali začetno besedo. Metoda, ki ne potrebuje celotne besede vnaprej, je Ukkonenov algoritem, ki priponsko drevo zgradi sprotno (angl. *on-line*), kot ga je izgradila tudi naivna metoda, ter potrebuje  $O(n)$  časa in prostora.

**Ukkonenov algoritem:** Ukkonenov algoritem deluje na podoben način kot prej predstavljena naivna metoda, saj dodaja v drevo črko po črko. Torej algoritem v  $i$ -tem koraku izgradi priponsko drevo, ki je lahko implicitno priponsko drevo, in predstavlja besedo  $T[1, i]$ . Algoritem v  $i$ -tem koraku doda v drevo črko  $x = T[i]$ . Naj bo  $\alpha$  niz, ki z  $x$  tvori niz  $T[a, i] = \alpha x$ , pri čemer je  $1 \leq a \leq i$  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 8a.
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 zadni 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 sredini 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 predhodnja povezava. Način je prikazan na Sliki 8b.
  - (b) Če pa obstaja taka pot, ki se nadaljuje po nizu  $\alpha$  z znakom  $x$ , se ne stori ničesar, saj je drevo že v implicitni obliki. Način je prikazan na Sliki 8c.

Te načine dodajanja se lahko vidi tudi na primeru izgradnje priponskega drevesa za besedo »KOKOŠ\$«. Postopek izgradnje z Ukkonenovim algoritmom in vmesna drevesa so prikazana na Sliki 9.

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). Vse točke, v katerih se bo v tem koraku zgodila razčlemba, imenujemo aktivne točke (angl. *active point*). Da bo algoritem učinkovit hranimo začetno aktivno točko in končno aktivno točko imenovano tudi končna točka (angl. *end point*), rezčlembe in dodajanja novih vozlišč v drevesu v trenutnem koraku. Na Sliki 9 je z rdečo označeno

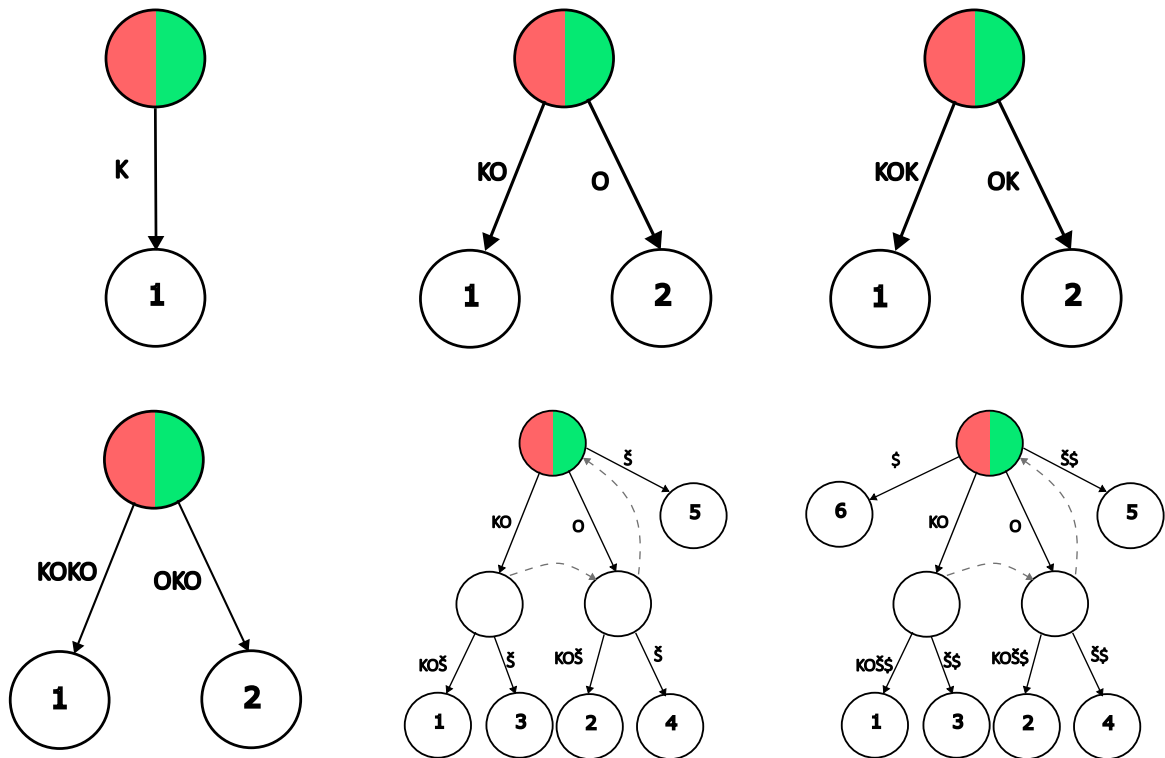


Slika 8: Načini dodajanja novih znakov v priponsko drevo. Na sliki kvadrati predstavljajo liste drevesa, krogi pa predstavljajo notranja vozlišča drevesa.

vozišče, ki se nahaja pred končno točko, ter z zeleno je označeno vozišče, ki pa se nahaja pred začetno točko. Premik po aktivnih točkah med začetno točko in končno točko poteka po priponskih povezavah. Pri tem pa algoritem v vsakem koraku poti izračuna, ali je že prišel v končno točko. Za to je potrebno hraniti zgolj trenutno aktivno točko ter zastavico, ki hrani vrednost, ali je ta točka tudi končna točka. Aktivna točka postane končna točka takrat, ko se način dodajanja spremeni iz 2a v 2b ali ko so vse implicitne pripone postale liste drevesa, saj od takrat ni več potrebno v trenutnem koraku izgradnje deliti povezave in ustavljati nova vozlišča. Pri tem velja tudi, da končna točka v koraku  $i - 1$  postane začetna aktivna točka v koraku  $i$ , saj se prva nova razčlenba v koraku  $i$  lahko zgodi zgolj v točki, kjer se je končala razčlenba v koraku  $i - 1$ , in to je končna točka koraka  $i - 1$ .

Vsako aktivno točko lahko predstavimo kot referenčni par  $(s, \alpha)$ , pri čemer je  $s$  vozišče pred aktivno točko in  $\alpha$  je niz iz vozišča  $s$  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 vozišče 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 9 v četrtem koraku izgradnje, 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 v  $(koren, (3, 4))$ , ki se bo razcepila in bo postala novo vozišče. Ta ista točka je lahko po koncu izgradnje 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 vozišče  $a$  otrok  $koren$ -a, na katerega kaže povezava z nizom »KO«.

Ukkonenov algoritem za izgradnjo priponskega drevesa, ki je predstavljen v [1], izgradi priponsko drevo s psevdokodo, ki je prikazana v Algoritmu 2. V algoritmu



Slika 9: Primer izgradnje priponskega drevesa z uporabo Ukkonenovaga algoritma za besedo »KOKOŠ\$«.

par  $(s, (k, i - 1))$  predstavlja kanonično obliko aktivnega vozlišča 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 2 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 niza ujemata, 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  $(false, v)$ .

**Algoritem 2:** Ukkonenov algoritem za izgradnjo priponskega drevesa**Vhod:** Beseda  $T$ , dolžine  $n$ **Izhod:** Priponsko drevo

```

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

```

**Izrek 3.7.** Ukkonenov algoritem zgradi priponsko drevo nad besedo  $T$  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 izgradnje dodaja nove povezave na poti iz končne točke  $kt_{i-1}$  koraka  $i - 1$  do končne točke  $kt_i$  koraka  $i$ , katera 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** obišče

$O(n)$  vozlišč skozi celotno izgradnjo 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 2. Ker se niz  $\beta$  poveča  $n$ -krat skozi celotno izgradnjo, potemtakem se tudi niz  $\beta$  lahko zmanjša največ  $n$ -krat skozi celotno izgradnjo. Torej funkcija `kanoničnaOblika` obišče največ  $n$  vozlišč v celotni izgradnji priponskega drevesa.

Zanka v vrstici 3 Algoritma 2 se izvede  $n$ -krat, medtem ko se zanka v vrstici 6 in funkcija `kanoničnaOblika` vsaka izvede v  $O(n)$  časa skozi celotno izgradnjo priponskega drevesa, iz česar sledi, da tudi izgradnja priponskega drevesa potrebuje  $O(n)$  časa, da se izvrši.

□

**Zaključek:** Tako McCreight algoritem [4] kot tudi Ukkonenov algoritem [1] zgradita priponsko drevo v času  $O(n)$ . Oba algoritma izdelata priponsko drevo, ki ima enake priponske povezave med vozlišči, pri tem pa se zalikujeta v umesnih drevesih. Za primerjavo umesnih dreves se lahko vzame Slika 7 in Slika 9, na katerih sta prikazani izgradnji priponskih dreves z obema algoritma za besedo »KOKOŠ\$«. V McCreightovem algoritmu vmesno drevo v  $i$ -tem koraku predstavlja  $i$  najdaljših pripon besedila. Pri čemer pa v Ukkonenovem algoritmu vmesno drevo v  $i$ -tem koraku predstavlja priponsko drevo besedila  $T[1, i]$ , ki je lahko bodisi implicitno bodisi eksplicitno predstavljeno, kar je posledica sprotne izgradnje priponskega drevesa.

Poleg linearne časovne zahtevnosti za izgradnjo priponskega drevesa, oba algoritma potrebujeta  $O(n)$  prostora za hrambo in gradnjo priponskega drevesa, saj ima vsako drevo  $n$  listov in največ  $n - 1$  notranjih vozlišč. Vsako vozlišče ima  $O(|\Sigma|)$  referenc, pri čemer je  $\Sigma$  abeceda vseh znakov uporabljenih v besedilu. Za daljša besedila je to lahko problem, saj velikost celotnega priponskega drevesa lahko presega velikost delovnega pomnilnika.

V nadaljevanju bo uporabljen Ukkonenov algoritem za izgradnjo priponskih dreves, ki bodo uporabljena pri empirični analizi v Poglavju 6.3, kjer bo tudi izmerjen vpliv pomanjkanja notranjega pomnilnika ter posledična uporaba zunanega pomnilnika (`Swap` razdelek na zunanjem spominu) namesto notranjega pomnilnika.

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

Osnovna poizvedba nad besedo  $T$  je  $\text{prisotnost}(T, P)$ , 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  $T$  natakoda tedaj, ko je  $P$  prisoten v  $T$ . Oziroma vzorec  $P[1, m]$  je prisoten v besedi  $T$  natakoda 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  $\text{prisotnost}(T, P)$  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 alociran 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  $\text{prisotnost}(T, P)$  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.

Za primer iskanja vzememo priponsko drevo na Sliki 4, 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 pred-

postavimo, 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 **koren.otroci['K']**  $\neq \text{NIL}$ . Ker **koren.otroci['K']** kaže na notranje vozlišče, ki ga imenujemo  $v$ , se lahko preveri, ali podniz na povezavi se ujema s  $P_1$  oziroma  $P_2$ . Ker se oba vzorca začneta s »KO« in se ujema s podnizom na povezavi, si zapomnemo, da smo pregledali dve črki, in nadaljujemo s pregledovanjem. Za vzorec  $P_1$  preverimo v vozlišču  $v$ , ali velja **v.otroci['Š']**  $\neq \text{NIL}$ . Ker obstaja taka povezava, lahko nadaljujemo z iskanjem, ampak smo že preverili vse črke vzorca, ker do vozlišča  $v$  sta bila že pregledana  $o = 2$  znaka in za iskanje naslednje povezave se je pregledal tudi znak  $P_1[3] = \text{Š}$ . Zato lahko trdimo, da vzorec  $P_1$  je prisoten v besedi »KOKOŠ«. Za razliko od vzorca  $P_1$  se za vzorec  $P_2$  v vozlišču  $v$  preveri, ali velja **v.otroci['T']**  $\neq \text{NIL}$ . Ker ne obstaja povezava, ki predstavlja podniz z prvim znakom 'T', torej tudi 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 *številoPonovitev*( $T, P$ ), 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 *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 *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 *številoPonovitev*( $T, P$ ) vrne število obiskanih listov v poddrevesu s korenem v  $v$ , če pa vzorec  $P$  ni prioten v besedi, pa vrne 0. Poizvedba *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 prvo potrebno preveriti, ali je vzorec prisoten v drevesu, je potrebno izvesti iste korake kot za poizvedbo *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, potemtakem za najti vse liste poddrevesa potrebujemo  $O(occ)$  časa.

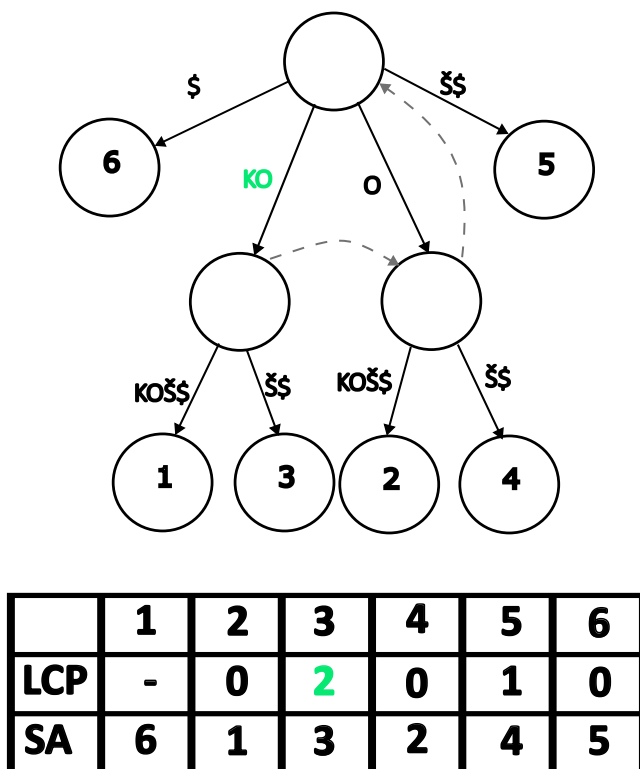
Za primer vzemimo besedo  $T = \text{»KOKOŠ«}$ , za katero imamo zgrajeno priponsko drevo, ki je prikazano na Sliki 4. V besedi želimo preveriti, kolikokrat se pojavi vzorec  $P = \text{»KO«}$  oziroma želimo narediti poizvedbo *številoPonovitev*( $T, P$ ). Poizvedbo



začnemo v *korenu* priponskega drevesa, kjer preverimo, ali obstaja povezava na vozlišče, ki se začne z znakom  $P[1] = 'K'$ . Ker obstaja povezava `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  $[1, 3]$ .

## 4 PRIPONSKO POLJE

Priponsko drevo nad besedo  $T$  dolžine  $n$  potrebuje za vsako vozlišče vsaj en kazalec ter dva »cela števila«, ki predstavljata začetek in konec predstavljenega podniza, zato se potrebuje vsaj  $6n$  »celih števil«. Pri tem se potrebuje še  $n$  dodatnih »celih števil« za shraniti priponske povezave, dodatnih  $2n$  »celih števil« za shraniti povezave na starše vozlišč ter še dodatnih  $n$  »celih števil« za shraniti indekse pripon, ki so shranjeni v listih. Torej priponsko drevo potrebuje med  $7n$  in  $10n$  »celih števil« za pravilno delovanje. Ker so pripone besedila indeksi v besedilu, ki predstavljajo indeks prvega znaka v priponi, se jih lahko shrani z  $n$  »celimi števili«. Če se indekse pripon shranjenih v polju in se indeksi uredijo v leksikografskem vrstnem redu pripon, se ta podatkovna struktura imenuje priponsko polje (angl. *Suffix array* oziroma SA).



Slika 10: Primer priponskega polja nad besedo »KOKOŠ\$«.

Priponsko polje je alternativni indeks besedila. Priponsko polje se uporablja namesto priponskega drevesa, ko se potrebuje prostorsko bolj učinkovito podatkovno strukturo in se lahko žrtvuje čas iskanja vzorcev v besedilu saj priponsko polje zasede

približno 8-krat manj prostora na delovnem pomnilniku kot priponsko drevo [24]. Priponsko polje si lahko predstavljamo kot polje listov priponskega drevesa, brez podatkov o notranjih vozliščih drevesa. To podobnost lahko vidimo tudi na Sliki 10, na kateri je prikazano priponsko drevo in priponsko polje za besedo »KOKOŠ\$«.

V nadaljevanju poglavja bodo predstavljena še podatkovna struktura, ki omogoča hitrejše iskanje v priponskem polju ter simuliranje priponskega drevesa. Zatem bodo predstavljeni še načini izgradnje priponskega polja ter implementacije poizvedb s pomočjo priponskega polja.

## 4.1 SKUPNA NAJDALJŠA PREDPONA

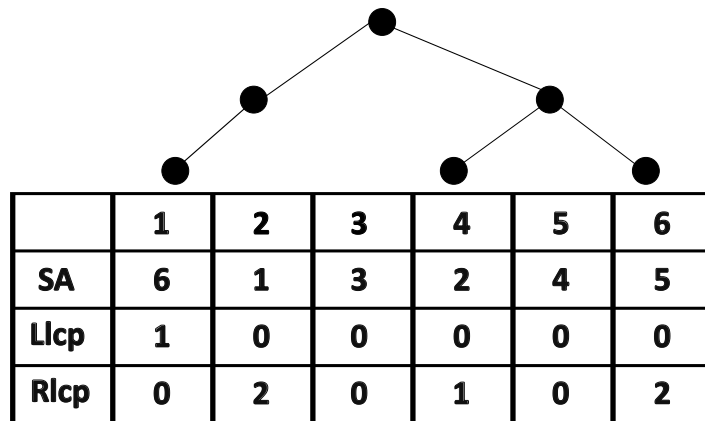
V priponskem polju so shranjene pripone besede  $T$ , ki so urejene v leksikografskem vrstnem redu, zato je najbolj učinkovita metoda iskanja bisekcija. Z bisekcijo je potrebnih  $O(\log n)$  primerjav med sredinsko pripono trenutnega intervala, na katerem se izvaja bisekcija, ter vzorcem  $P$ , torej je potrebno  $O(m \log n)$  časa za preveriti, ali je vzorec  $P$  prisoten v besedi  $T$ . 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. V tem podpoglavju bo predstavljena predstavljen podatkovna struktura  $LCP$  polje. V podpoglavju 4.3 pa bo predstavljena uporaba te podatkovne strukture za iskanje po priponskem polju.

Polje najdaljših skupnih predpon (angl. *Longest common prefix* oziroma  $LCP$ ) hrani v vsaki celici

$$LCP[i][j] = lcp(T[SA[i], n], T[SA[j], n]),$$

kar je dolžina predpone, ki je skupna obema priponama. Za tako definirano  $LCP$  polje je potrebno  $O(n^2)$  prostora. Pri tem velika večina celic ne bo nikoli uporabljena pri bisekciji. V vsakem koraku bisekcije se preverja, ali je vzorec večji od sredinske točke  $M$  na intervalu  $[L, R]$ . Torej za vsak možen interval bisekcije 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 bisekciji, potem potrebujemo dve  $LCP$  polji, in sicer prvega za shraniti dolžina najdaljše predpone med  $M$  in  $L$ , ki ga imenujemo  $L-LCP$ , in drugega za shraniti dolžina najdaljše predpone med  $M$  in  $R$ , ki ga imenujemo  $R-LCP$ . Primer teh dveh polj je prikazan na Sliki 11, na kateri je prikazano tudi drevo sredinskih točk bisekcije [24].

Prostorska zahtevnost te implementacije  $LCP$  polja je  $O(n)$ . Če želimo biti bolj natančni  $L-LCP$  in  $R-LCP$  polji hranita vsak  $n$  »celih števil«, torej skupaj s priponskim poljem potrebujejo  $3n$  »celih števil«, kaj je še vedno 2- do 5-krat manj prostora kot ga potrebuje ekvivalentno priponsko priponsko drevo.

Slika 11: Primer  $L$ -LCP in  $R$ -LCP polji za priponskega polja nad besedo »KOKOŠ\$«.

#### 4.1.1 Simulacija priponskega drevesa

V priponskem drevesu je dolžina najdaljše skupne predpone dveh pripon dolžina podniza najglobljšega skupnega predhodnika (angl. *Lowest common ancestor* oziroma LCA) obeh listov, ki predstavljata priponi. Ampak trenutna implementacija  $LCP$ -polja je namenjena pospešitvi iskanja po priponskem polju in uporablja  $L$ -LCP in  $R$ -LCP polji. Zato se potrebuje bolj splošno  $LCP$ -polje, ki bi nadomestilo  $L$ -LCP in  $R$ -LCP polji in bi omogočalo simuliranje priponskega drevesa.

Vrednost v  $L$ -LCP[ $M$ ] predstavlja dolžino najdaljše skupne predpone  $lcp(T[SA[M], n], T[SA[L], n])$ , pri čemer je  $L$  začetni indeks intervala bisekcije s sredinsko točko v  $M$ , in vrednost funkcije  $lcp$  označimo kot  $k$ . Vemo tudi, da je  $SA[L] < SA[M]$ , torej imajo vse pripone na intervalu med  $L$  in  $M$  paroma najdaljšo skupno predpono dolžine vsaj  $k$ , saj vse pripone na tem intervalu so leksikografsko večje od  $SA[L]$  in manjše od  $SA[M]$ . To pomeni, da za vsak  $i$ , ki je  $L < i \leq M$ , velja  $k \leq lcp(T[SA[i-1], n], T[SA[i], n])$ . Posledično obstaja tak  $i$ , za katerega velja  $lcp(T[SA[i-1], n], T[SA[i], n]) = k$ . 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  $LCP[2, n]$ , za katerega velja

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

Primer takega  $LCP$  polja je prikazan na Sliki 10. Na sliki je označena z zeleno barvo vrednost  $LCP[3]$ , ki je najdaljša skupna predpona med priponama  $SA[2]$  in  $SA[3]$ . Ta vrednost je tudi označena na priponskem drevesu na sliki [14, 26].

Novo  $LCP$  polje potrebuje zgolj  $O(n)$  »celih števil«. Pri tem pa potrebuje dodatno podatkovno strukturo za učinkovito iskanje najmanjše vrednosti na intervalu oziroma  $rmq$ , ki je potrebna za nadomestiti  $L$ -LCP in  $R$ -LCP polji. 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 prepo-

časno, saj potrebuje iskanje z  $L$ -LCP in  $R$ -LCP polji  $O(m + \log n)$  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  $O(m + \log n)$ .

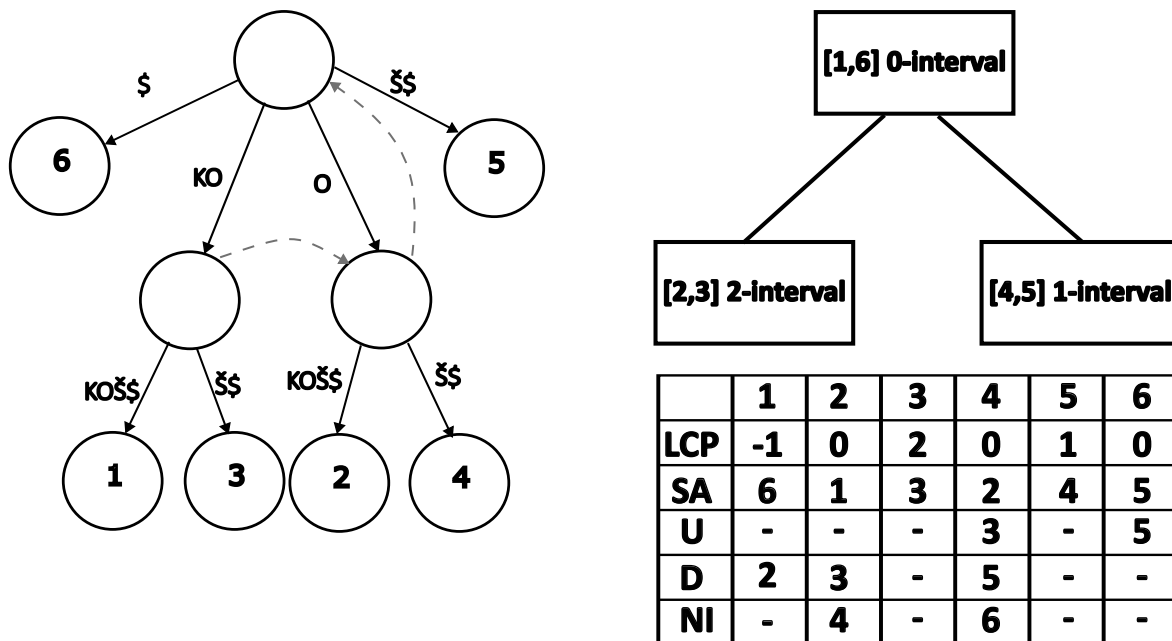
Posplošeno LCP polje se tudi lahko uporabi simuliranje priponskega drevesa. Vrednost  $LCP[i]$  predstavlja črkovno dolžino vozlišča  $v$ , za katerega velja  $v = LCA(l_{i-1}, l_i)$ , pri čemer  $l_i$  predstavlja pripono  $SA[i]$  in  $l_{i-1}$  predstavlja pripono  $SA[i - 1]$ . Kasai idr. [14] so 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  $v$ . Ker je priponsko polje ekvivalentno listom priponskega drevesa, so vsi listi poddrevesa s korenem v  $v$  ekvivalentni intervalu priponskega polja  $SA[L, R]$ , pri čemer  $SA[L]$ -ta pripona je predstavljena s skrajno levim listom in  $SA[R]$ -ta pripona pa je predstavljena s skrajno desnim listom v poddrevesu. Torej so vse vrednosti intervala  $LCP[L + 1, R]$  vsaj  $l$ . Pri tem velja tudi, da pripona  $SA[L - 1]$  in pripona  $SA[R + 1]$  se zagotovo razlikujeta s priponami  $SA[L]$  in  $SA[R]$  vsaj v  $l$ -tem znaku, sicer bi bile v poddrevesu. Torej velja  $LCP[L] < l$  in  $LCP[R + 1] < l$ . Torej lahko imenujem tak interval  $[L, R]$   $l$ -interval, ki je definiran s sledečo definicijo [26].

**Definicija 4.1.** Interval  $[i, j]$  je  $l$ -interval v LCP polju, natanko tedaj ko velja:

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

Koren drevesa predstavlja prazen niz in njegovo poddrevo je celotno drevo, zato koren predstavlja 0-interval  $[1, n]$ . Ker pa je LCP polje definirano zgolj za indekse od 2 do  $n$ , se lahko definira indeks 1 kot  $LCP[1] = -1$ . Ta popravek ne vpliva na pravilnost delovanja priponskega polja, saj ne obstaja pripona  $SA[0]$ . Znotrja vsakega  $l$ -intervala  $[L, R]$  lahko obstajajo drugi  $l'$ -intervali, za katere velja  $l < l'$ . Takih  $l'$ -intervalov je največ  $n_l + 1 = O(|\Sigma|)$ , pri čemer je  $n_l$  število ponovitev vrednosti  $l$  v LCP polju na intervalu  $[L, R]$ . Torej lahko predstavimo relacijo med  $l$ -intervali kot LCP intervalno drevo (angl. *LCP interval tree*). Primer LCP intervalnega drevesa



Slika 12: Primer intervalnega drevesa nad priponskim poljem in LCP poljem besede »KOKOŠŠ« ter njegova predstavitev s tabelo. Na sliki je dodano tudi priponsko drevo besede »KOKOŠŠ«.

je prikazan na Sliki 12. Prikazana je tudi relacija med priponskim drevesom in *LCP* intervalnim drevesom. [26].

Za učinkovito simulacijo priponskega drevesa je potrebno učinkovito shraniti *LCP* intervalno drevo. Najbolj intuitiven način za shraniti topologijo *LCP* intervalnega drevesa shrani za vsak interval polje  $D$  (angl. *down* oziroma dol), ki hrani indekse začetkov podintervalov vsebovanih znotraj intervala. Pri tem je potrebno  $O(|\Sigma|n)$  prostora za shraniti vseh  $O(n)$  polj  $D$ . Prostorsko zahtevnost se lahko zniža na  $O(n)$  »celih števil«, tako da vsak interval hrani vrednost  $D$ , ki kaže na začetek prvega podintervala, ter verednost  $NI$  (angl. *next index* oziroma naslednji indeks ali naslednji brat), ki pa hrani začetni indeks naslednjega podintervala mojega starša oziroma naslednjega brata v *LCP* intervalnem drevesu. Vsako vozlišče hrani dve »celi števili«, zato potrebujemo  $O(n)$  prostora v  $O(\log n)$  poljih (eno polje za vsako globino). Ker potrebujemo približno  $2n$  »celi števili« se pojavi vprašanje, ali je možno predstaviti ta števila z dvema poljema  $D$  in  $NI$  dolžine  $n$ . Problem pri takem zapisu je razločevanje med začetkom intervala starša in začetkom intervala prvega otroka, česar ni mogoče storiti, torej ni mogoče predstaviti zapisa samo s poljema  $D$  in  $NI$ .

To pa se lahko stori s tremi dodatnimi polji »celih števil« dolžine  $n$ , torej potrebujemo  $5n$  »celih števil« za simulirati priponsko drevo z *LCP* intervali. Pri tem pa se potrebuje med 1,4- in 2-krat manj prostora kot ekvivalentno priponsko drevo. V polju  $D$  se na mestu  $i$  shranjeni začetek drugega intervala (drugega sina) od najdalj-

šega intervala (najbolj plitvega vozlišča), ki se začne na mestu  $i$ , oziroma  $D[i]$  hrani največji indeks  $j$  večji od  $i$ , za katerega velja  $LCP[j] > LCP[i]$  in vse vrednosti v  $LCP$  med indeksi  $i$  in  $j$  so večje od  $LCP[j]$ . V polju  $NI$  na mestu  $i$  so shranjeni začetki naslednjega intervala, ki ima istega starša kot interval z začetkom v  $i$ , oziroma  $NI[i]$  hrani prvi indeks  $j$  večji od  $i$ , za katerega velja  $LCP[j] = LCP[i]$  in vse vrednosti v  $LCP$  med indeksi  $i$  in  $j$  so večje od  $LCP[j]$ . S polji  $D$  in  $NI$  lahko iščemo po intervalih, ki se ne začnejo na istem mestu kot njihovi starši. Če se želimo sprehoditi po prvem podintervalu, pa potrebujemo dodatno polje, ki hrani informacijo o otrocih prvega podintervala. To polje imenujemo  $U$  (angl. *up* oziroma *gor*) in na  $i$ -tem mestu hrani začetek drugega podinterval predhodnega brata, ki se je končal na mestu  $i - 1$ . Z drugimi besedami  $U[i]$  hrani položaj prvega indeksa  $j$  manjšega od  $i$ , za katerega velja  $LCP[j] > LCP[i]$  in vse vrednosti v  $LCP$  med indeksi  $j$  in  $i$  so večje ali enake od  $LCP[j]$ . Torej za interval  $[i, j]$  velja  $D[i] = U[j + 1]$ . Začetek intervala drugega otroka v prvem podintevalu intervala  $[i, j]$  izračunamo kot  $U[D[i]]$ . S temi polji se lahko sprehodimo po priponskem polju, kot bi se sprehodili po priponskem drevesu [26]. Na Sliki 12 je prikazan tudi zapis intervalnega drevesa s polji  $U$ ,  $D$  in  $NI$ .

**Poizvedbe z simulacijo:** Poizvedba se začne z indeksom  $i = 1$  in  $j = n$  in preverili smo prvih  $o = 0$  znakov. Nato povečamo vrednost  $o$  za ena in najdemo interval, ki se začne z vrednostjo  $P[o]$ , z Algoritmom 3. Nato v vsakem koraku iskanja za trenutni interval  $[i, j]$  poiščemo vrednost  $l = \min\{LCP[k]; i < k \leq j\}$ . Vemo tudi, da se prvih  $o$  znakov  $P$ -ja ujema s priponami na intervalu  $[i, j]$ . Nato preverimo, ali je  $P[o, k] = T[SA[i] + o, SA[i] + k]$ , pri čemer je  $k = \min\{l, m\}$ . Če se ujemata, popravimo  $o \leftarrow k$  in začnemo iskati podinterval  $[i', j']$ , za katerega velja  $P[o + 1] = T[SA[i'] + o + 1]$ , ki bo postal novi interval  $[i, j]$ . Če je  $o = m$ , lahko vrnemo, da je  $P$  prisoten v  $T$ , poročamo, da se  $P$  ponovi  $(j - i + 1)$ -krat v  $T$ , ali pa izgradimo seznam indeksov besede  $[SA[i], \dots, SA[j]]$ , odvisno od poizvedbe. Če pa se  $P[o, k]$  ne ujema s  $T[SA[i] + c, SA[i] + k]$  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  $P[o + 1]$ , ter 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  $[i', j']$  je prikazan na Algoritmu 3 in ne potrebuje  $LCP$  polja, saj smo predhodno že naračunali vrednost  $l$ , ko se je preverjalo prisotnost podniza vzorca  $P[o, k]$ .

Torej iskanje s simulacijo priponskega drevesa 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

---

**Algoritem 3:** Algoritem za iskanje pod intervala

---

**Vhod:** Začetek intervala  $i$ , konec intervala  $j$ , znak  $c$ , število pregledanih znakov  $l$ **Izhod:** Interval  $[i', j']$ 

```

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

```

---

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 interval, pri čemer konec intervala je 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 pole.

**Opazka:** 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 vsaki  $i$  med 2 in  $n$  vrednosti  $LCP[i] = lcp(LA[i - 1], LA[i])$  in vrednost  $LCP[1] = -1$ .



## 4.2 IZGRADNJA

Podobno kot priponsko drevo, je mogoče izgraditi 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 uporabi priponsko drevo za izgraditi priponsko polje. 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 izgrajeno),
2. sprehod v globino po drevesu in zapis indeksov pripon iz listov v polje v vrstnem redu obiska.

V koraku sprehoda je mogoče izgraditi 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 do takrat je sprehod poteka od lista navzgor in se je v vozlišču obrnil ter poteka od vozlišča do lista navzdol.

Opisan algoritem izgradi 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 5-krat več dodatnega prostora, kot ga zasede priponsko polje, in razlog za izgradnjo priponskega polje je zmanjšanje količine spomina, ki ga zasede indeks besede. Zato bi potrebovali algoritem, ki izgradi priponsko polje v  $O(n)$  časa in ne izgradi priponskega drevesa.

**Izgradnja s urejanjem pripon:** Priponsko polje je polje indeksov pripon urejenih v leksikografskem vrstnem redu, zato se lahko uporabi urejanje za izgradnjo priponskega polja. 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 koren 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 »celih števil« in 2 bitni polji dolžine  $n$  [24].

Pri tem pa obstaja bolj učinkovit algoritem za igraditi priponskega pola, ki so ga predlagali Ko idr. [25] in potrebuje  $O(n)$  časa. Ideja algoritma temelji na deljenju pripon na  $L$  pripone (pripona  $T[i, n]$  je manjša od pripone  $T[i + 1, n]$ ) in  $S$  pripone (pripona  $T[i, n]$  je večja od pripone  $T[i + 1, n]$ ), pri čemer 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 izgradi priponsko polje v štirih korakih: deljenje pripon na  $S$  in  $L$ , ureditev polja glede na prvi znak pripone, premik  $L$  pripon na začetek veder s sprotnim urejanjem  $S$  pripon in dodatno urejanje  $L$  pripon s premikom pripone na začetek intervala, če je pripona, ki je za en znak daljša, tudi  $L$  pripona. Vsak korak je lahko storjen z enim sprehodom po polju oziroma besedi, za kar je potrebno  $O(n)$  časa. Podobno kot algoritem s korenskim urejanjem, je prostorska zahtevnost tega algoritma tudi  $O(n)$ , saj se potrebuje 3 polja »celih števil« in 3 dodatna bitna polja (dva dolžine  $n$  in enega dolžine  $n/2$ ) [25].

## 4.3 POIZVEDBE

Priponsko polje je bilo izgrajeno kot alternativa priponskemu drevesu, ki potrebuje manj prostora. Zato se ga uporablja za indeksiranje besede  $T$  in posledično iskanje vzorcev v njej. V tem podpoglavju bodo predstavljene implementacije za priponsko polje istih poizvedb, ki so bile predstavljene za priponsko drevo.

Najbolj osnovna poizvedba, ki bo uporabljena kot osnova za ostali dve poizvedbi, je  $prisotnost(T, P)$ . Prisotnost vzorca  $P$  dolžine  $m$  v besedi  $T$  lahko preverimo z bisekcijo, saj so pripone v priponskem polju urejene. V vsakem koraku bisekcije preverimo, ali se pripona na sredini interval  $[L, R]$  (v prvem koraku je  $L = 1$  in  $R = n$ ) ujema z vzorcem  $P$ , pri čemer označimo indeks te pripone kot  $M$ . Če je  $T[SA[M], SA[M] + m - 1] = P$ , 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  $R - L > 1$ , ko je  $R - L = 1$  in  $P[k] \neq T[SA[M] + k]$  potem  $P$  in prisoten v besedi  $T$ . Tako opisan

postopek potrebuje  $O(m \log n)$  časa, saj bisekcija potrebuje  $O(\log n)$  primerjav, vsaka primerjava pa potrebuje dodatnih  $O(m)$  primerjav, da ugotovimo, ali se pripona in vzorec ujemata.

V predhodnem podpoglavju je bila predstavljena podatkovna struktura  $LCP$  polje, ki se jo lahko uporabi za pospešiti iskanje, in sicer implementacija s polji  $L-LCP$  in  $R-LCP$ . Poizvedba z  $LCP$  polji še vedno temelji na bisekciji, saj je bisekcija najbolj učinkovit način iskanja po urejenem polju. V osnovni različici poizvedbe v vsakem koraku bisekcije preverimo, ali je vzorec manjši, večji ali enak srednji priponi intervala  $[L, R]$  z indeksom  $M$ . Pri tem lahko preštejemo, koliko znakov se ujema med pripono  $SA[M]$  in  $P$  ter označimo to vrednost s  $k$ . To znanje lahko uporabimo za znižati število primerjav črk med  $P$  in  $SA[M]$  na  $O(m)$  skozi celotno izvajanje bisekcije. Torej na začetku vsakega koraka vemo, da se  $P$  ujema z vsaj  $k$  znaki in če smo v levem oziroma desnem podintervalom 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  $L-LCP[M]$  in pripona  $SA[L]$  je leksikografsko manjša od  $SA[M]$ . Torej obstajajo tri možnosti:

1.  $k < L-LCP[M]$ , 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 sredne predhodnega intervala, po tem pomeni, da je  $P[k+1] > T[SA[L+k+1]] = T[SA[M+k+1]]$ . Torej naslednji pregledani interval je  $[M, R]$ .
2.  $k > L-LCP[M]$ , potem se  $SA[L]$ -ta pripona in  $SA[M]$ -ta pripona manj ujemata kot  $SA[L]$ -ta pripona in  $P$  in označimo  $L-LCP[M] = l$ . 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  $SA[M]$ -ta pripona. Torej se bisekcija nadaljuje na intervalu  $[L, M]$ .
3.  $k = L-LCP[M]$ , potem je potrebno preveriti, ali je  $P$  večji od  $SA[M]$ -te pripone. Pri tem ni potrebno preveriti prvih  $k$  znakov, saj vemo, da se ujemajo, ker je se  $SA[L]$ -ta pripona ujema s  $SA[M]$ -ta pripona v  $k$  znakih in se  $SA[L]$ -ta pripona tudi ujema z  $P$  v  $k$  znakih. Med preverjanjem štejemo, koliko znakov se ujema in to zabeležimo kot  $k'$ . Če je  $P[k+k'+1] < T[SA[M]+k+k'+1]$  potem nadaljujemo v interval  $[L, M]$ , sicer je  $P[k+k'+1] > T[SA[M]+k+k'+1]$  in nadaljujemo v interval  $[M, R]$ . Preden nadaljujemo v naslednji interval popravimo  $k \leftarrow k + k'$ , pri čemer  $k \leq m$ .

Simetrično velja tudi, če smo v levem podintervalu, pri tem pa uporabimo  $R-LCP$ , saj je  $R$  predhodna srednja točka intervala.

Poizvedba  $prisotnost(T, P)$  potrebuje  $O(m + \log n)$  časa, da preveri prisotnost vzorca v besedi. Pri tem potrebujemo  $O(\log n)$  primerjav, saj se uporablja bisekcijo za učinkovito iskanje po urejenem polju. Skozi celotno izvajanje poizvedbe pa se

potrebuje dodatnega  $O(m)$  časa za primerjave med znaki vzorca  $P$  in zanki pripon na sredini intervalov bisekcije.

Idejo poizvedbe  $prisotnost(T, P)$  lahko uporabimo za implementacijo poizvedbe  $številoPonovitev(T, P)$ . Poizvedba vrne število  $occ$ , ki je število ponovitev vzorca  $P$  v besedi  $T$ . Naivna implementacija bi bila štetje pripon levo in desno od pripone  $SA[M]$ , ki je prva pripona v bisekciji, ki se ujema z  $P$ . Ta postopek potrebuje  $O(m + \log n + occ)$  časa, pri čemer se lahko uporabi  $LCP$  polje za iskanje ponovitev, saj sta vrednosti  $LCP[M]$  in  $LCP[M + 1]$  večji ali enaki  $m$  natanko tedaj, ko se priponi  $SA[M - 1]$  in  $SA[M + 1]$  začneta z vzorcem  $P$ . Poizvedbo se lahko pohitri na  $O(m + \log n)$  z dvema bisekcijama. Prva bisekcija je potrebna za iskanje začetka intervala  $L$  vseh pripon, ki se začnejo z  $P$ , druga bisekcija pa je potrebna za iskanje konca intervala  $R$  vseh takih pripon. Število pripon, ki se začne s  $P$ , je  $occ = R - L + 1$ . Vsaka bisekcija 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, ki se začnejo s  $P$ ,  $[L, R]$  za izdelati seznam indeksov ponovitev vzorca  $P$  v besedi  $T$ . Za najti interval  $[L, R]$  je potrebno  $O(m + \log n)$  časa, za pretveriti interval v priponskem polju v seznam indeksov pripon pa je potrebnih dodatnih  $occ$  dostopov do priponskega polja oziroma  $O(occ)$  časa. Torej poizvedba  $seznamPojavov(T, P)$  potrebuje  $O(m + \log n + occ)$  časa.

## 5 KOMPAKTNA PREDSTAVITEV PRIPONSKEGA DREVEŠA

Za daljše besede  $T$  lahko priponska drevesa presežejo velikost notranjega pomnilnika. Prva možna rešitev tega problema je uporaba priponskega polja ter  $LCP$  polja za simulirat priponsko drevo, ampak se pri 1,4-krat daljši vhodni besedi ponovno pojavi problem preseženja velikosti notranjega pomnilnika. Torej se potrebuje rešitev, ki bo še bolj prostorsko učinkovita.

Vsako vozlišče priponskega drevesa hrani niz, ki predstavlja vhodno povezavo, in reference na otroke, starša ter na vozlišče, na katerega kaže priponska povezava. Zato prostor, ki ga zasedejo drevesa na pomnilniku, ni odvisna zgolj od števila vozlišč, ampak je odvisna 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 (pri tem je všteti v abecedo tudi, znak za konec besedila), potrebuje med 60 in 144 GB notranjega pomnilnika, ki je odvisno od podatkov shranjenih v vozliščih [13]. Ekvivalentno priponsko polje pa potrebuje med 12 in 60 GB notranjega pomnilnika odvisno od dodatnih podatkovnih struktur, ki so bile dodane priponskemu polju. Zato prostorsko učinkovita rešitev ne sme temeljiti na referencah pomnilniških naslovov.

Ta problem se da rešiti s kompaktno predstavitevjo podatkovnih struktur. Podatkovna struktura, ki se jo bo predstavilo v tem poglavju, je imenovana kompaktno priponsko drevo (angl. *Compressed Suffix Tree* oziroma CST) in predstavlja eno možno kompaktno predstavitev priponskega drevesa. Preden se lahko definira kompaktno predstavitev je potrebno definirati abstraktno podatkovno strukturo priponsko drevo, ki poda vse operacije, ki so potrebne za pravilno delovanje priponskega drevesa [2].

**Definicija 5.1.** Abstraktna podatkovna struktura priponsko drevo nad besedilo  $T$  podpira sledeče operacije:

1.  $koren()$ : vrne koren priponskega drevesa,
2.  $jeList(v)$ : vrne »Da«, če je vozlišče list, sicer vrne »Ne«,
3.  $otrok(v, z)$ : vrne otroka  $w$ , 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 od vozlišča  $v$ ,
6.  $pbrat(v)$ : vrne vozlišče  $w$ , ki je predhodni brat od vozlišča  $v$ ,
7.  $starš(v)$ : vrne vozlišče  $w$ , ki je starš od vozlišča  $v$ ,
8.  $povezava(v, i)$ : vrne  $i$ -to črko na povezavi do vozlišča  $v$ ,
9.  $globinaNiza(v)$  vrne število znakov na poti iz korena do vozlišča  $v$ ,
10.  $lca(v, w)$ : vrne najnižjega skupnega prednika od  $v$  in  $w$ ,
11.  $sl(v)$ : vrne vozlišče  $w$ , na katerega kaže priponska povezava iz vozlišča  $v$ .

V podpoglavju 4.1.1 je bil predstavljen način simulacije priponskega drevesa s pomočjo priponskega polja  $SA$  in polja najdaljših skupnih predpon  $LCP$ . Z uporabo kompaktnih različic teh dveh podatkovnih struktur je možno implementirati kompaktno priponsko drevo. Ker pa 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. Torej sledi definicija za podatkovno strukturo kompaktno priponsko drevo.

**Definicija 5.2.** Kompaktno priponsko drevo nad besedilom  $T$  je sestavljeno iz sledečih podatkovnih struktur:

1. topologija drevesa  $\tau$ ,
2. kompaktno priponsko polje  $SA$ ,
3. kompaktna predstavitev polja najdaljših skupnih predpon  $LCP$ .

V nadaljevanju poglavja bo predstavljena Sadakanejeva [2] implementacija kompaktne priponskega drevesa, ki je prva kompaktna predstavitev priponskega drevesa. V Poglavju 2.3 je bilo predstavljenih več različnih predstavitev topologije dreves. Implementacija kompaktne priponskega drevesa pa uporablja predstavitev z zaporedjem uravnoteženih oklepajev.

**Topologija drevesa:** Ker se za kompaktno priponsko drevo uporablja BP kot topologija 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 so potrebne dve dodatni podatkovni strukturi za operacijo  $rang$  ( $rang_0$ ,  $rang_{01}$ ) in tri za operacijo  $izbira$  ( $izbira_0$ ,  $izbira_1$  in  $izbira_{01}$ ). Ostale operacije pa uporabljajo podatkovno strukturo za višek predstavljeno, ki sta jo predlagala Munro in Raman [15] in je implementirana na podoben način kot dodatna podatkovna struktura za  $rang$ , namesto  $rmM$ -drevesa. To

podatkovno strukturo imenujemo  $L$ , ki hrani polje najnižjih viškov  $L'$  in v celici  $L'[i]$  je shranjen 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 ji omogoča konstanten čas izvajanja  $rmq$  poizvedbe nad viškom. Ta podatkovna struktura potrebuje  $o(n)$  dodatnih bitov. In sicer se izgradi dvodimenzionalno celico tabele  $M[i][k]$ , ki shrani položaj najmanjšega viška na intervalu  $L'[i..i + 2^k - 1]$ . Torej operacija  $lca(v, w)$  se izračuna kot

$$\min \left\{ \begin{array}{l} \min\{L[v, e]\}, \\ \min\{M[v'][k], M[w' - 2^k + 1][k]\}, \\ \min\{L[s, w]\} \end{array} \right\},$$

pri čemer je  $k = \lfloor \log(w' - v') \rfloor$ ,  $v'$  predstavlja vedro v  $L'$ , ki je prvo vedro po vedru z vozliščem  $v$ ,  $w'$  je vedro v  $L'$ , ki je prvo vedro pred vedru 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 konstantno časa [2, 3].

V primeru, da so potrebne dodatne operacije, ki temeljijo na  $rmq$  operaciji in  $|\Sigma| > \log n$ , se lahko doda še  $rmM$ -drevo, ki omogoča izvedbo operaciji 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 besedilom  $T$  dolžine  $n$  potrebuje  $4n + o(n)$  bitov.*

*Dokaz.* Priponsko drevo nad besedilom  $T$  ima  $2n - 1$  vozlišč: od tega je  $n - 1$  notranjih vozlišč in  $n$  listov. Zato je potrebnih  $4n - 2$  bitov za zapis topologije drevesa  $\tau$  s sekvenco uravnoveščenih oklepajev. Torej celotna podatkovna struktura topologije drevesa potrebuje  $4n + o(n)$  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 kompaktnega 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 imenovan  $FM$ -indeks pa uporablja  $LF$  funkcijo, ki je inverzna funkcija od  $\Psi$  [2, 6].

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

**Definicija 5.4.** Kompaktno priponsko polje nad besedilom  $T$ , ki je uporabljeno za predstavitev kompaktne priponskega drevesa, podpira sledeče operacije:

1.  $pripona(i)$  vrne  $SA[i]$  v času  $t_{SA}$ ,
2.  $inverz(i)$  vrne  $j = SA^{-1}[i]$ , 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 izgradnjo priponskih povezav v času  $t_{\Psi}$ , čeprav ni potrebna za pravilno delovanje priponskega polja. Iz definicije funkcije  $\Psi$  je razvidno, da jo lahko simuliramo z uporabo operacij  $inverz(i)$  in  $pripona(i)$ . Iz Definicije 5.4 sledi, da mora kompaktno priponsko polje podpirati funkcijo  $\Psi$ , zato implementacije z  $FM$ -indeksom ne pridejo v poštev.

V nadaljevanju tega dela poglavja bo predstavljena preprosta različica kompaktne 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 potrebno hraniti besedila  $T$  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 kompaktne zapisa polja  $\Psi$  temelji na dejstvu, da se lahko premikamo po besedilu 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 bitno polje sprememb v prvem tanku  $D$ , pri čemer  $D[i] = 1$ , ko je  $i = 1$  ali  $T[SA[i]] \neq T[SA[i + 1]]$ , ter s poljem znakov  $S$  urejenih 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  $C$  začetkov intervalov v  $\Psi$  polju, ki se začnejo z poljubnim zankom  $c \in \Sigma$ , oziroma  $C[c] = i$  za poljubni 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  $\Psi$  polje z 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 znaka  $c$  v besedilu  $T$ . Torej velja  $\Psi_c[i'] = izbira_1(B_c, i')$  ali  $\Psi[i] = izbira_1(B_c, i - C[c])$ , kjer  $c = S[rang_1(D, i)]$ . Bitna polja  $B_c$  so zelo redka (število enic je bistveno manjše kot število 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 ohranimomo 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 besedila  $T$ , ter potrebuje  $t_{\psi} = O(1)$  časa [6].



Do sedaj predstavljena podatkovna struktura omogoča zgolj iskanje števila ponovitev vzorca v besedilu, saj omogoča premikanje med intervali z priponami z istim začetnim zankom, ne pa lokacije pojavov vzorca v besedilu. Za to sta potrebni dodatni podatkovni strukturi, ki nadomestita priponsko polje  $SA$  ter inverzno polje  $SA^{-1}$ . Oba polja se lahko nadomesti s poljema vzorcev. Priponsko polje  $SA$  se vzorči  $l = \Theta(\log n)$ -krat, kar je dober kopromis med časovno in postorsko zahtevnostjo. Pri tem se uporabi 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 izračuna kot  $SA[i] = SA_S[rang_1(B, i_k)] - k$ , pri čemer je  $i_k$   $k$ -kratna aplikacija 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 se ta vzorči na enakomernih intervalih dolžine  $l = \Theta(\log n)$ , kar je dober kopromis med časovno in postorsko zahtevnostjo. Torej ima  $i$ -ta celica polja  $SA_S^{-1}[1, \lfloor n/l \rfloor]$  vrednost  $SA_S^{-1}[i] = SA^{-1}[il]$ . Najprej se izračuna  $i' = \lfloor i/l \rfloor l$ , nato se  $i - i'$ -krat aplicira funkcija  $\Psi$  nad vrednostjo  $j' = SA_S^{-1}[i'/l]$ , kar se zapiše kot  $SA^{-1}[i] = \Psi^{i-i'}[j'] = \Psi^{i-i'}[SA_S^{-1}[i'/l]]$ . Isti razmislek kot za polje  $SA$  velja tudi za polje  $SA^{-1}$ , 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 lahko vzorči z večjim faktorjem  $l$  kot se je vzorčilo priponsko polje  $SA$ . Na ta način se zmanjša prostorsko zahtevnost, ampak se poslabša časovne zahtevnosti operacije, ki se jo poredko uporablja [6].

Tako predstavljeno kompaktno priponsko polje potrebuje  $|\Sigma|n + O(n)$  bitov pred stiskanjem zloredekih bitnih polji. Ker pa je večina bitnih polji redkih, saj je število bitov z vrednostjo 0 veliko večje kot število bitov z vrednostjo 1, se lahko binta polja zakodira na bolj kompakten način z enim od algorimom stiskanja. Torje stisnjena bitna polja potrebujejo  $nH_0(T) + O(n + |\Sigma|w)$  bitov za implementacijo kompaktnega priponskega polja. Potrebni čas za izračun funkcije  $\Psi$  je  $t_\Psi = O(1)$  ter potrebni čas za iskanje po priponskem polju in inverznem priponskem polju je  $t_{SA} = lt_\Psi = t_\Psi O(\log n) = O(\log n)$ .

**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  $LCP[i] = PLCP[SA[i]]$  ter jo je lažje predstaviti v kompaktnem zapisu [2, 6].

Permutacijo  $PLCP[1, n-1]$  je lažje kompaktno predstaviti, saj je zaporedje  $PLCP[i] + 2i$  za vsak  $i$  med 1 in  $n - 1$  strogo naraščajoče. Zaporedje je strogo naraščajoče, saj je  $PLCP[i + 1] \geq PLCP[i] - 1$ . To lahko enostavno dokažemo, saj če je  $PLCP[i] = 0$

mora biti tudi  $PLCP[i+1] = 0$ . Drugače pa obstajata  $T[j, n]$  in  $T[i, n]$  ter  $T[i, n] < T[j, n]$ , ki imata najdaljšo skupno predpono dolžine  $PLCP[j] > 0$ . Potem ima  $T[i+1, n]$  najdaljšo skupno predpono s  $T[j+1, n]$  dolžine  $PLCP[j] - 1$  in vse pripone, ki so leksikografsko manjši od  $T[j+1, n]$  in imajo skupno predpono, imajo najdaljšo skupno predpono s  $T[j+1, n]$  vsaj dolžino  $PLCP[j] - 1$ . Torej je  $PLCP[j+1] = PLCP[j] - 1$  in poslednično še vedno velja  $PLCP[j] + 2j < PLCP[j+1] + 2(j+1) = PLCP[j] + 2j + 1$ . Ker je  $PLCP[n-1] + 2(n-1) < 2n$ , saj ima lahko  $T[n-1, n]$  dolžino najdaljše skupne predpone s poljubno pripono 1, se lahko permutacijo  $PLCP$  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 je izračunana z uporabo formule  $LCP[i] = izbira_1(H, SA[i]) - 2SA[i]$ . Poizvedba potrebuje  $O(t_{SA})$  časa, saj je potrebno izračunati vrednost  $SA[i]$ , ki se jo izračuna v  $O(t_{SA})$  časa, ostale operacije pa potrebujejo konstanten čas. Če je uporabljena predhono predstavljena implementacija kompaktne polja je čas poizvedbe v  $LCP$  polj enak  $O(\log n)$ . Prostorska zahtevnost polja  $LCP$  je predstavljena s sledečo lemo:

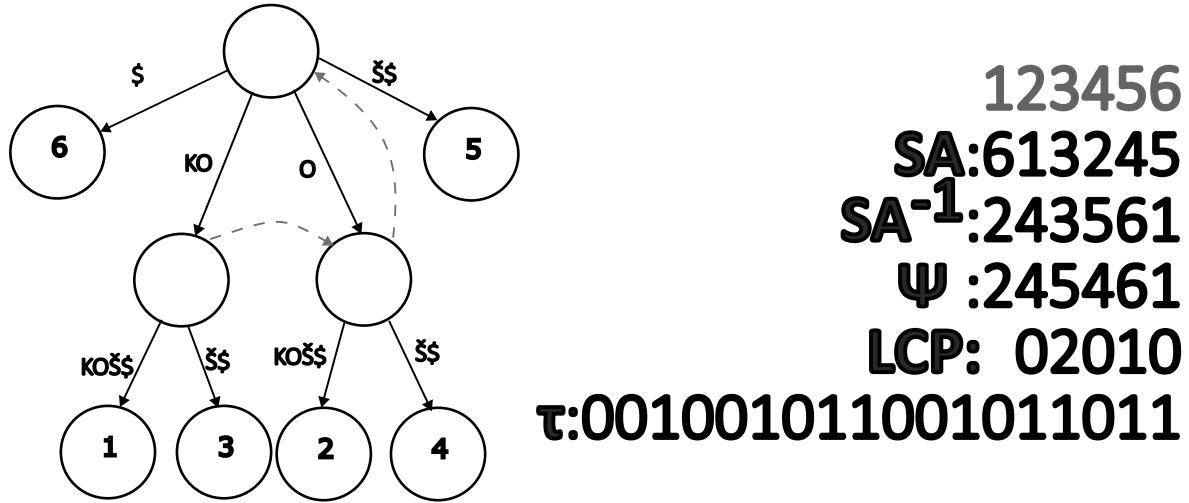
**Lema 5.5.** *Podatkovna struktura  $LCP$  potrebuje  $2n + o(n)$  bitov za pravilno delovanje.*

*Dokaz.* Podatkovna struktura  $LCP$  je shranjena kot bitno polje  $H[1, 2n-1]$ . Bitno polje  $H$  potrebuje dodatnih  $o(n)$  bitov za dodatno 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 za shraniti polje  $LCP$  potrebuje  $2n + o(n)$  bitov.  $\square$

Sedaj, ko so bile predstavljene vse podatkovne strukture, ki sestavljajo kompaktno priponsko drevo, je možno 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  $LCP$ .



Slika 13: Primer priponskega drevesa (levo) in kompaktnega priponskega drevesa (desno) za besedo »KOKOŠŠ«.

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

Obstaja več različnih implementacij kompaktnega priponskega polja, zato Sada-kane [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.** *Kompaktno priponsko drevo implementirano s pomočjo kompaktnega priponskega polja, ki potrebuje  $|CSA| = nH_h + O(n \log \log n / \log_{|\Sigma|} n)$  bitov, 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 ga sta ga predlagala Grossi in Vitter [16].

**Posledica 5.8.** *Kompaktno priponsko drevo implementirano s pomočjo kompaktnega priponskega polja, ki potrebuje  $O(\frac{1}{\epsilon} n \log |\Sigma|)$  bitov, potrebuje  $|CST| = O(\frac{1}{\epsilon} n \log |\Sigma|)$  bitov. Pri tem je  $\epsilon$  poljubna konstanta, ki ima vrednost  $0 < \epsilon < 1$ .*

Primer kompaktnega priponskega drevesa je prikazan na Sliki 13. Zaradi lažje berljivosti so vse komponente kompaktnega priponskega drevesa (na desni strani slike) prikazane v ne kompaktni obliki. Priponsko polje je prikazano v treh vrsticah (vrstica, ki se začne s  $SA$ , vrstica, ki se začne s  $SA^{-1}$ , in vrstica ki se začne s  $\Psi$ ).

**Izboljšave:** Prostorsko zahtevnost kompaktnega priponskega drevesa je možno 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  $\delta = \omega(\log_{|\Sigma|} n)$  in predstavlja faktor vzorčenja. Vzorčeno drevo potrebuje  $o(n) =$

$O(n/\log_{|\Sigma|} n)$  bitov. Poleg vzorčenja topologije drevesa, so se znebili tudi *LCP* polja. To jim je omogočalo znižanje prostorske zahtevnosti iz  $|CSA| + 6n + o(n)$  bitov na  $|CSA| + o(n)$  bitov. Na ta račun pa se je dvignila časovna zahtevnost nekaterih operacij za  $\omega(\log n)$  krat, če se želi ohraniti  $o(n)$  dodatnega prostora. Nekatere od teh operacij so predhodno potrebovale konstanten čas izvajanja.

Preden se nadaljuje z izgradnjo *CST* in poizvedbami, se lahko nadaljuje s primerom človeškega genoma iz začetka poglavja. S priponskim drevesom se potrebuje 144 GB notranjega spomina za indeksirati celoten človeški genom. Z uporabo kompaktne priponskega drevesa, namesto priponskega drevesa, pa je potrebnih približno 3 GB delovnega spomina za indeksirati celoten genom. Razlika v zahtevanem prostoru omogoča, da je lahko celotno kompaktno priponsko drevo shranjeno v delovnem spominu, za razliko od priponskega drevesa, za katerega to ni mogoče<sup>1</sup>.

## 5.1 IZGRADNJA

Kompaktno priponsko drevo je možno izgraditi iz priponskega drevesa. Tak način izgradnje kompaktne priponskega drevesa ni prostorsko učinkovit, saj je potrebno prvo izgraditi celotno priponsko drevo. Zato bi bilo bolje izgraditi kompaktno priponsko drevo neposredno iz besedila, kot je to storjeno za priponsko drevo.

Kompaktno priponsko drevo je mogoče izgraditi neposredno iz vhodnega besedila  $T$ . Pri tem so potrebne dodatne podatkovne strukture za izgradnjo posamičnih komponent. Tudi te dodatne podatkovne strukture so kompaktne podatkovne strukture. Izgradnja kompaktne priponskega drevesa poteka v sledečih treh korakih:

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

Prva zgrajena podatkovna struktura je kompaktno priponsko polje  $SA$ . Ker je priponsko polje implementirano s pomočjo funkcije  $\Psi$  (shranjena v istoimenskem polju), je potrebno naračunati polje  $\Psi$ . To je storjeno s pomočjo Burrows–Wheelerjeve preslikave (angl. *Burrows–Wheeler Transform* oziroma BWT), ki je izdelana s pomočjo priponskega polja in besedila. Preslikavo se gradi iz leve proti desni, torej se lahko BWT izgradi postopoma z deli priponskega polja dolžine  $b$ , saj za  $b = n/\log n$  je potrebnih  $o(n)$  bitov dodatnega prostora in  $O(n \log n)$  časa [6].

---

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

Priponsko polje  $SA$  se ne razdeli na  $b$  delov, ampak je lažje najti najmanjše število predpon, ki se pojavijo na začetku največ  $b$ -tim priponam, ter se na ta način razdeli priponsko polje  $SA$ . Sledi izračun pripon, ki sodijo v določeno vedro dolžine  $b$  priponskega polja  $SA$ . Pripone v tem vedru se leksikografsko uredijo, najpogosteje z uporabo korenskega urejanja (angl. *RadixSort*). Vrednost  $BWT$  se naračuna s pregledom vseh pripon v vedru. Pripone se shrani v polje  $L[1, n]$ , kjer je  $L[kb + j] = T[A[j] - 1]$ , pri čemer  $k$  predstavlja zaporedno število vedra in  $T[0] = \$$  [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  $L[i] = c$  potem se nastavi  $B_c[i] = 1$ . To je lahko storjeno brez dodatnega prostora za polje  $L$ , saj se ne zapiše črke  $c = T[A[j] - 1]$  v polju  $L$ , ampak se jo lahko neposredno zapiše v bitno polje  $B_c[kb + j] = 1$  [6].

Čas potreben za izgradnjo kompaktnega 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 naračunat vse pripone.

Bitno polje  $H$ , ki predstavlja  $LCP$  polje, je naslednja izgrajena 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 izračunan s štetjem zaporednih znakov, ki se ujemajo med  $T[1, n]$  in  $T[SA[SA^{-1}[1] - 1], n]$ . Ker je  $PLCP[i - 1] \leq PLCP[i] + 1$ , potem je vrednost  $PLCP[i]$  število skupnih zaporednih znakov med  $T[i + d, n]$  in  $T[SA[SA^{-1}[i] - 1] + d, n]$ , pri čemer  $d = \max\{PLCP[i - 1] - 1, 0\}$ . Pri izgradnji bitnega polja  $H$  ni potrebo prvo naračunati polje  $PLCP$ , saj število lahko takoj zapiše enico na mesto  $PLCP[i] + 2i$  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 potemtakem je za izgradnjo potrebno  $O(nt_{SA})$  ali za predhodno predstavljeno implementacijo  $O(n \log n)$ .

Zadnja izgrajena podatkovna struktura je topologija priponskega drevesa  $\tau$ . Čeprav se zdi, da ni mogoče izgraditi topologije drevesa, ne da bi se izgradilo celotno priponsko drevo, je to mogoče zgraditi zgolj z uporabo priponskega polja  $SA$  in  $LCP$  polja. Ideja algoritma za izgradnjo topologija priponskega drevesa  $\tau$  je podobna algoritmu Kasai idr. [14].

To je storjeno s prehodom po priponskem polju  $SA$  iz leve proti desni ter za vsako pripono  $i$  se doda nov list. Novi dodani list  $i$  je novo skrajno desno vozlišče v do sedaj izgrajenem drevesu ter ima skupnega predhodnika  $v$  z  $i - 1$ -im listom na besedni 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 izgradi priponsko drevo zgolj iz priponskega polja  $SA$  in  $LCP$  polja. Predstavljena metoda je implementirana z uporabo sklada, ki hrani vozlišče  $v$  ter besedno globino  $sd(v)$ . Na skladu se ne shrani kazalca 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 se pridemo vozlišče, za katerega pogoj drži. Tako vozlišče vedno obstaja, saj ima koren besedno 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 sprazni 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  $O(n \log n)$ . V vsakem koraku je v topologijo drevesa dodan nov list. Za vsak list je lahko dodano novo vozlišče na sklad, če za besedno 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  $O(n \log n)$  potrebni čas izgradnje topologije drevesa  $\tau$ .

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

*Dokaz.* Kompaktno priponsko drevo je sestavljeno iz treh delov, torej je potrebno analizirati časovno zahtevnost izgradnje vsakega dela. Za izgradnjo kompaktnega 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 potrebo izračunati dolžino predpone, v kateri se ujema s predhodno pripono. Pri tem je potreben za vsako pripono en dostop to priponskega polja, ki traja  $O(\log n)$  časa, 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.

Torej skupni čas za izgradnjo kompaktnega priponskega drevesa iz besedila  $T$  je  $O(n \log n) + O(n \log n) + O(n \log n) = O(n \log n)$ .  $\square$

## 5.2 POIZVEDBE

Naslednji razdelek v Tabeli 4 predstavlja osnovne poizvedbe, ki se izvajajo nad priponskimi drevesi. Najbolj preprosta poizvedba med njimi je *prisotnost(vzorec)*, ki preveri, če je vzorec *vzorec* dolžine *m* prisoten v besedilu *T*. To poizvedbo se lahko izvede direktno nad besedilom z uporabo *KMP* algoritma, ki potrebuje  $O(n+m)$  časa, kar je enako kot čas potreben za izgraditi priponsko drevo,  $O(n)$ , in preveriti prisotnost vzorca v drevesu,  $O(m)$ . Ko želimo preveriti prisotnost večjega števila vzorcev v besedilu, uporabimo priponsko drevo. Prisotnost vzorca v besedilu s priponskim drevesom se preveri s prehodom iz korena proti listom drevesa. Pri tem se preveri, ali se nizi na povezavah ujemajo z iskanim vzorcem. V vsakem novem vozlišču je potrebno najti otroka, ki se začne z naslednjim znakom v vzorcu, za kar je potrebno  $O(\log |\Sigma|) = O(1)$ , saj je velikost abecede konstantna. Potrebno je še preveriti, ali je celoten vzorec *vzorec* prisoten v priponskem drevesu, kar ima časovno zahtevnost  $O(m \log |\Sigma|) = O(m)$ .

---

**Algoritem 4:** 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
3      če s > e potem
4          return [-1, -1]
5      c = P[i]
6      [s', e'] = [rang1(Bc, s - 1) + 1, rang1(Bc, e)]
7      [s, e] = [C[c] + s', C[c] + e']
8  return [s, e]
```

---

V kompaktnem priponskem drevesu se prisotnost vzorca išče s pomočjo vzvratnega iskanja (angl. *Backward Search*) vzorca v priponskem polju *SA*. Vzvratno iskanje za predhodno predstavljeno kompaktno priponsko polje, ki je prikazano s Algoritmom 6, potrebuje  $O(mt_\Psi)$  časa, da se izvrši. Za drugačno implementacijo kompaktne priponskega polja, se nadomesti vrstico 6 v Algoritmu 6 z binarnim iskanjem nad  $\Psi_c$  in zato je potrebno  $O(m \log nt_\Psi)$  časa. Nato pa je potrebo preveriti, ali je  $[s, e] \neq [-1, -1]$ , za kar je potrebno konstantno časa. Torej je potrebno  $O(mt_\Psi)$  časa za preveriti prisotnost vzorca ali  $O(m \log nt_\Psi)$  z uporabo binarnega iskanja. V primeru, da se uporabi opisano kompaktno priponsko polje, pa se poizvedba izvrši v času  $O(m)$ .

Naslednja poizvedba je *številoPonovitev(vzorec)*, ki vrne število pojavov vzorca v besedilu, kar je ekvivalentno številu pripon v priponskem drevesu, ki se začnejo z

vzorcem *vzorec*. V priponskem drevesu je potrebno najti vozlišče  $v$ , za katerega velja  $\text{besedilo}(v)[1, m] = \text{vzorec}$ . Po tem takem je število ponovitev vzorca enako številu listov v poddrevesu s korenem v vozlišču  $v$ . Štetje vseh listov zahteva  $O(n)$  časa, za iskanje vozlišča  $v$  pa je potrebno  $O(m)$  časa, saj iskanje poteka na isti način, kot v poizvedbi  $\text{prisotnost}(\text{vzorec})$ . V kompaktnem priponskem drevesu, pa je operacija ponovno implementirana s pomočjo vzvratnega iskanja. Operacija  $\text{številoPonovitev}(\text{vzorec})$  je implementiran kot razlika  $e - s$ , kjer  $s$  predstavlja prvo pripono, ki se začne z vzorcem *vzorec*, in  $e$  je zadnja tako pripona, torej je razlika  $s - e$  število pripon, ki se začnejo z vzorcem *vzorec*. Torej operacija ponovno potrebuje  $O(mt_\Psi)$  oziroma  $O(mt_\Psi \log n)$  časa, da se izvrši.

Zadnja predstavljena poizvedba pa je  $\text{seznamPojavov}(\text{vzorec})$ , ki vrne vsa začetna mesta pojavov vzorca *vzorec* v besedilu  $T$ . To je ekvivalentno seznamu vseh pripon besedila  $T$ , ki se začnejo z vzorcem *vzorec*. V priponskem drevesu je to implementirano na podoben način, kot poizvedba  $\text{številoPonovitev}(\text{vzorec})$ , pri čemer namesto štetja listov v poddrevesu s korenem v vozlišču  $v$ , se v seznam pripon dodaja vse pripone, ki so predstavljene kot listi v poddrevesu s korenem v vozlišču  $v$ . V kompaktnem priponskem drevesu pa je poizvedba ponovno implementirana s pomočjo vzvratnega iskanja. Z vzratnim iskanjem se naračuna interval v priponskem polju  $SA[s, e]$ . Za pridobitev položajev ponovitev vzorca v besedilu, je potrebno ustvariti seznam  $[SA[s], SA[s+1], \dots, SA[e]]$ , kar zahteva še dodatnih  $e - s$  korakov, pri čemer vsak korak potrebuje  $O(t_{SA})$  časa za se izvršit. Torej poizvedba  $\text{seznamPojavov}(\text{vzorec})$  potrebuje  $O(mt_\Psi + t_{SA})$  oziroma  $O(mt_\Psi \log n + t_{SA})$  časa. Z predhodno predstavljeno implementacijo kompaktnega priponskega drevesa pa je potrebno  $O(m + \log n)$  časa.

Iz implementacij poizvedb nad kompaktnim priponskim drevesom se lahko vidi, da so vse tri poizvedbe implementirane zgolj s pomočjo kompaktnega priponskega polja. Iz tega se lahko sklepa, da sta topologija drevesa  $\tau$  in  $LCP$  polje odvečni podatkovni strukturi. To je res zgolj za osnovne poizvedbe nad besedilom  $T$ , ki so lahko izvršene zgolj s priponskim polje v enakem času. Poizvedbe, kot so najdaljši ponavljajoči podniz, najdaljši palindrom, ki je implementirana s pomočjo priponskega drevesa konkatencije obrata  $T'_z$  besedila  $T_z$ ,  $T = T_z \# T'_z \$$ , in najdaljši skupni niz besedila  $T_1$  in  $T_2$ , ki je implementirana s pomočjo priponskega drevesa konkatencije obeh besedil  $T = T_1 \# T_2 \$$ . Na primer poizvedbe najdaljši ponavljajoči podniz je podniz  $T[SA[i], SA[i] + \text{globinaNiza}(v)]$ , pri čemer  $i$  je 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  $\text{globinaNiza}(v)$  ni potrebno naračunati, saj je enaka  $LCP[i]$ , torej se še vedno izvede  $O(nt_{SA})$  časa, pri tem pa ni uporabljena topologija drevesa. V primeru, da želimo najti drugi najdaljši ponavljajoč se podniz  $T[SA[j], SA[j] + \text{globinaNiza}(u)]$ , pri čemer je  $j$  skrajno levi list poddrevesa s korenem v notranjem vozlišču  $u = sl(v)$ . Za izračunat le tega pa je potrebo  $O(nt_{SA} + t_\Psi)$  časa ter se uporabi



vse tri podatkovne strukture kompaktnega priponskega drevesa [3, 5, 6].

## 6 OPERACIJE NAD PRIPOSKIMI DREVESMI

V tem poglavju bodo analizirane implementacije operacij nad priponskimi drevesi iz Definicije 5.1, časovna zahtevnost izgradnje priponskega drevesa in prostorska zahtevnost priponskega drevesa ter kako je implementirano iskanje vzorcev s pomočjo priponskih dreves.

Poglavje je razdeljeno na tri dele. V prvem delu so predstavljene teoretične razlike med implementacijami (različne implementacije kompaktne priponskega polja ter implementacija priponskega drevesa). V drugem delu je predstavljena metoda empiričnega testiranja med kompaktnim priponskim drevesom in priponskim drevesom. V zadnjem delu pa so predstavljeni rezultati empirične primerjave.

### 6.1 TEORETIČNA ANALIZA

Prostorska zahtevnost kompaktne priponskega drevesa je nižja (velikost kompaktne priponskega drevesa je  $|CSA| + 6n + o(n)$  bitov za razliko od  $O(n)$  kazalcev oziroma  $O(n \log n)$  bitov priponskega drevesa), kar je tudi vidno iz primera človeškega genoma, vendar imajo nekatere operacije na račun kompaktne predstavitve višjo časovno zahtevnost. V Tabeli 4 so prikazane razlike: v prostorski zahtevnosti priponskega drevesa in kompaktne priponskega drevesa ter v časovni zahtevnosti operacij priponskega drevesa, v časovni zahtevnosti različnih poizvedb nad priponskim drevesom in v časovni zahtevnosti izgradnje drevesa. Tabela 4 je razdeljena na tri dele: v prvem delu so zbrane operacije priponskega drevesa, v drugem delu so zbrane poizvedbe nad priponskim drevesom, v tretjem delu tabele pa sta zbrani časovna zahtevnost izgradnje ter prostorska zahtevnost.

Kompaktno priponsko drevo je lahko implementirano s pomočjo različnih različic kompaktne priponskega polja, zato imajo operacije, ki so implementirane s pomočjo priponskega polja, različne časovne zahtevnosti. V Tabeli 4 imajo zato nekatere operacije časovno zahtevnost  $O(t_\Psi)$  (časovna zahtevnost funkcije  $\Psi$ ) ali  $O(t_{SA})$  (časovna zahtevnost dostopa do priponskega polja) ter prostorska zahtevnost kompaktne priponskega drevesa vsebuje člen  $|CSA|$ . Posledica 5.7 in Posledica 5.8 predstavljata prostorsko zahtevnost kompaktne priponskega drevesa z uporabo dveh implementacij kompaktne priponskega polja, zato so v Tabeli 5 predstavljene prostorske in

Tabela 4: Časovna zahtevnost operacij priponskega drevesa, izgradnje priponskega drevesa in iskanja v priponskem drevesu ter prostorska zahtevnost priponskega drevesa.

	$ST$	$CST$
$koren()$	$O(1)$	$O(1)$
$jeList(v)$	$O(1)$	$O(1)$
$otrok(v, z)$	med $O(1)$ in $O( \Sigma )$	$O(\log  \Sigma  t_{SA})$
$prviOtrok(v)$	$O(1)$	$O(1)$
$starš(v)$	$O(1)$	$O(1)$
$nbrat(v)$	$O(1)$	$O(1)$
$pbrat(v)$	$O(1)$	$O(1)$
$povezava(v, i)$	$O(1)$	$O(t_{SA})$
$globinaNiza(v)$	$O(1)$	$O(t_{SA})$
$lca(v, w)$	$O(1)$	$O(1)$
$sl(v)$	$O(1)$	$O(t_{\Psi})$
$številoPonovitev(vzorec)$	$O(m + occ)$	$O(mt_{\Psi})$
$seznamPojavov(vzorec)$	$O(n + occ)$	$O(mt_{\Psi} + occ \cdot t_{SA})$
$prisotnost(vzorec)$	$O(m)$	$O(mt_{\Psi})$
$izgradnja(T)$	$O(n)$	$O(nt_{SA} + n \log n)$
$velikost$	$O(n)$ kazalcev	$ CSA  + 6n + o(n)$ bitov

časovne zahtevnosti obeh implementacij priponskega polja.

V nadaljevanju poglavja bodo bolj natančno primerjane različne implementacije operacij priponskega drevesa ter poizvedbe nad priponskimi drevesi. Prostorska zahtevnost ter časovna zahtevnost izgradnje ne bo podrobno primerjana, saj so bile podrobno predstavljene v predhodnih poglavjih. Čeprav večina operacij nad priponskim drevesom ohrani enak čas izvajanja tudi v kompaktnem priponskem drevesu, se implementacija teh spremeni. Implementacije bodo predstavljene v vrstnem redu, tako kot se pojavijo v Tabeli 4.

Tabela 5: Primerjava prostorske zahtevnosti ter časovne zahtevnosti različnih implementacij kompaktne priponskega polja.

Implementacija	$ CSA $ [bit]	$t_{\Psi}$	$t_{SA}$
Grossi idr. [17]	$nH_h + O(n \log \log n / \log_{ \Sigma } n)$	$O(\log  \Sigma )$	$O(\log^2 n / \log \log n)$
Grossi in Vitter [16]	$O(\frac{1}{\epsilon} n \log  \Sigma )$	$O(1)$	$O(\log^{\epsilon} n)$

Prva primerjava implementacije je storjena za operacijo  $koren()$ , ki vrne koren priponskega drevesa. Operacija v obeh implementacijah potrebuje konstantni čas, da se izvrši. V priponskem drevesu operacija  $koren()$  vrne kazalec na vozlišče koren. V kompaktnem priponskem drevesu pa operacija  $koren()$  vrne število 1, ki predstavlja položaj začetka korena oziroma uklepaja, ki predstavlja koren, v topologiji drevesa  $\tau$ . Iz operacije  $koren()$  je razvidno, da čeprav obe implementaciji potrebujeta konstantni čas, da se izvršita, sta implementirani na dva popolnoma različna načina ter vrneta dve popolnoma različni vrednosti.

Naslednja primerjava implementacij bo narejena za operacijo  $jeList(v)$ , ki preveri ali je vozlišče  $v$  list priponskega drevesa ali ne. V priponskem drevesu je implementirana operacija  $jeList(v)$ , tako da se preveri ali za vsak znak iz abecede  $\Sigma$  obstaja povezava do drugega vozlišča. Če ne obstaja nobena taka povezava, potem je vozlišče  $v$  list, sicer ni. V kompaktnem priponskem drevesu pa se preveri, če topologija drevesa  $\tau[v, v + 1] == 01$ , saj ima list  $v$  predstavitev drevesa z uravnoteženimi oklepaji obliko 01. Obe implementaciji sta prikazani v Algoritmu 5, in sicer je v zgornjem delu prikazana implementacija za priponsko drevo, v spodnjem delu pa implementacija za kompaktno priponsko drevo.

---

**Algoritem 5:** Implementacija operacije  $jeList(v)$  za ST in CST

---

**Vhod:** Vozlišče  $v$

```

1  (/* Implementacija za ST                                     */)
2  za  $i = 1, \dots, |\Sigma|$ 
3  |   če  $i$ -ti otrok obstaja potem
4  |   |   return False
5  return True
6  (/* Implementacija za CST                                     */)
7  če  $\tau[v] == 0 \wedge \tau[v + 1] == 1$  potem
8  |   return True
9  return False

```

---

Operacija  $otrok(v, z)$  vrne vozlišče  $u$ , katerega povezava, ki kaže nanj iz vozlišča  $v$ , se začne z znakom  $z \in \Sigma$ . Operacija je implementirana v priponskih drevesih s pomočjo dvojiškega iskanja, saj so povezave med vozliščem  $v$  in njegovimi otroki urejene v leksikografskem vrstnem redu. Torej je potrebnih  $O(\log |\Sigma|)$  primerjav. Podobno je operacija implementirana tudi v kompaktnem priponskem drevesu. Pri tem pa je potrebno dostopati do priponskega polja za preveriti znak. Znak se izračuna s sledečo formulo:

$$\begin{aligned}
a &= \text{besedilo}(u)[\text{globinaNiza}(v) + 1] = \\
&= S[\text{rang}_1(D, SA^{-1}[SA[\text{rangList}(u)] + \text{globinaNiza}(v)])],
\end{aligned}$$

pri čemer dodatna operacija  $\text{besedilo}(v)$  vrne besedilo, ki ga predstavljajo nizi na povezavah na poti iz korena do vozlišča  $v$ . Če  $a = z$ , potem se vrne vozlišče  $u$ , sicer se nadaljuje z iskanjem. Ker sta v enačbi dva dostopa do priponskega polja in en klic operacije  $\text{globinaNiza}(v)$ , potrebuje vsaka primerjava  $O(t_{SA})$  časa, da se izvrši, zato potrebuje celotna operacija  $O(t_{SA} \log |\Sigma|)$  časa, da se izvrši. Pri tem se lahko opazi, da je operacija  $\text{otrok}(v, z)$  prva operacija, za katero se časovni zahtevnosti razlikujeta.

Za razliko od operacije  $\text{otrok}(v, z)$ , se operacija  $\text{prviOtrok}(v)$ , ki vrne skrajno levega otroka od  $v$ , v obeh implementacija izvede v konstantnem času. V priponskem drevesu operacija  $\text{prviOtrok}(v)$  vrne kazalec na prvo vozlišče v seznamu otrok vozlišča  $v$ , ki se lahko s psevdokodo napiše kot `v.otroci[1]`. V kompaktnem priponskem drevesu pa operacija vrne število  $v + 1$ , ki predstavlja položaj začetka prvega otroka oziroma uklepaj, ki predstavlja prvega otroka, v topologiji drevesa  $\tau$ .

Operacija  $\text{starš}(v)$  vrne vozlišče  $u$ , ki je starš od vozlišča  $v$ . Operacijo je mogoče implementirati v konstantnem času bodisi za priponsko drevo bodisi za kompaktno priponsko drevo. V priponskem drevesu je operacija implementirana, kot kazalec v vozlišču  $v$ , ki kaže nazaj na vozlišče  $u$ . To je lahko zapisano s psevdokodo kot `v.starš`. V kompaktnem priponskem drevesu pa je operacija  $\text{starš}(v)$  implementiran s pomočjo dejstva, da uklepaj in zakepaj, ki predstavljata vozlišče  $u$ , oklepata celotno poddrevo s korenem  $u$ . Torej  $u$ , ki je starš od  $v$ , mora oklepati  $v$ . Torej je operacija  $\text{starš}(v)$  implementirana kot  $u = \text{oklepa}(\tau, v)$ . Operacijo  $\text{oklepa}$  je možno implementirati v času  $O(1)$  s pomočjo predhodno predstavljene dodatne podatkovne strukture ali v času  $O(\log n)$  s pomočjo  $rmM$ -drevesa.

Operacija  $\text{starš}(v)$  se uporabi pri implementaciji operacij  $\text{nbrat}(v)$  in  $\text{pbrat}(v)$ , saj sta vozlišča  $v$  in  $u$  brata, natanko tedaj ko velja  $\text{starš}(v) = \text{starš}(u)$ .

Operacija  $\text{nbrat}(v)$ , ki vrne vozlišče  $u$ , ki je desni brat od vozlišča  $v$ , in potrebuje za obe implementaciji priponskega drevesa  $O(1)$  časa, da se izvrši. V priponskem drevesu vrne kazalec na vozlišče  $u$ , za katerega velja, da če je vozlišče  $v$  na  $i$ -tem mestu v seznamu otrok, potem je vozlišče  $u$  na  $i + 1$ -tem mestu. To se lahko zapiše s psevdokodo kot `v.starš.otroci[i+1]`. V primeru, da se zaporedno število vozlišča  $v$  v seznamu otrok ne beleži, je lahko le to naračunano v času  $O(\log |\Sigma|)$ , kar še vedno omogoča iskanje v konstantnem času. V kompaktnem priponskem drevesu pa je naslednji brat od  $v$  predstavljen kot prvi uklepaj po zaklepaju od vozlišča  $v$ , kar se lahko zapiše kot  $u = \text{zapri}(\tau, v) + 1$ . Operacijo  $\text{zapri}$  je možno implementirati v času  $O(1)$  s pomočjo predhodno predstavljene dodatne podatkovne strukture ali v času  $O(\log n)$  s pomočjo  $rmM$ -drevesa.

Podobno kot operacija  $nbrat(v)$ , tudi operacija  $pbrat(v)$ , ki vrne vozlišče  $u$ , ki je levi brat od vozlišča  $v$ , potrebuje za oba tipa priponskega drevesa  $O(1)$  časa, da se izvrši. V priponskem drevesu vrne kazalec na vozlišče  $u$ , za katerega velja, da če je vozlišče  $v$  na  $i$ -tem mest v seznamu otrok, potem je vozlišče  $u$  na  $i - 1$ -vem mestu. To se lahko zapiše s psevdokodo kot  $\mathbf{v.starš.otroci}[i-1]$ . V primeru, da se zaporedno število vozlišča  $v$  v seznamu otrok ne beleži, je lahko le to naračunano v času  $O(\log |\Sigma|)$ , kar še vedno omogoča iskanje v konstantnem času. V kompaktnem priponskem drevesu pa je naslednji brat od  $v$  predstavljen kot prvi uklepaj, ki odpre oklepaj levo od uklepaja od vozlišča  $v$ , kar se lahko zapiše kot  $u = odpri(\tau, v - 1)$ . Operacijo  $odpri$  je možno implementirati v času  $O(1)$  s pomočjo predhodno predstavljene dodatne podatkovne strukture ali v času  $O(\log n)$  s pomočjo  $rmM$ -drevesa.

Naslednja operacija, ki se ji teoretični čas izvajanja poslabša z uporabo kompaktnih priponskih dreves, je  $povezava(v, i)$ , ki vrne  $i$ -ti znak povezave, ki vodi v vozlišče  $v$ . V priponskih drevesih je niz na povezavi predstavljen kot par indeksov  $(s, e)$ , kjer  $s$  predstavlja začetek prvega pojava niza v besedilu  $T$ ,  $e$  pa predstavlja konec niza v besedilu. Torej je  $i$ -ti znak v nizu  $z = T[s + i - 1]$ , če velja  $0 < i \leq e - s$ . Za to je potrebno  $O(1)$  časa. Operacija  $povezava(v, i)$  pa je v kompaktnih priponskih drevesih implementirana na podoben način, kot je operacija  $otorok(v, z)$ . Implementirana je s sledečo formulo:

$$\begin{aligned} z &= besedilo(v)[globinaNiza(starš(v)) + i] = \\ &= S[rang_1(D, SA^{-1}[SA[rangList(u)] + globinaNiza(starš(v))] + i - 1)]. \end{aligned}$$

Operacija dvakrat dostopa do priponskega polja, ki za vsak dostop potrebuje  $O(t_{SA})$  časa, ter potrebuje rezultat operacije  $globinaNiza(u)$ , pri čemer  $u = starš(v)$ , ki tudi potrebuje  $O(t_{SA})$  časa, da se izvrši, po tem takem potrebuje operacija  $povezava(v, i)$  tudi  $O(t_{SA})$  časa, da se izvrši.

Operacija  $globinaNiza(v)$  vrne število znakov na povezavah na poti iz korena proti vozlišču  $v$ . V priponskem drevesu je možno beležiti za vsako vozlišče njegovo besedno globino. Torej je možno operacijo implementirati s časovno zahtevnostjo  $O(1)$ . V kompaktnem priponskem drevesu je vrednost  $LCP[i]$ , kjer  $i$  je zaporedno število pripone, ki ga predstavlja skrajno desni list v poddrevesu, ki ima kot koren drugega otroka od  $v$ . To je možno, saj ima vsako notranje vozlišče vedno vsaj dva otroka. To se izračuna z uporabo sledeče formule

$$sd(v) = LCP[i] = izbira_1(H, SA[i]) - 2SA[i],$$

pri čemer je  $i = rang_{01}(\tau, nbrat(prviOtrok(v)))$ . To velja zgolj za notranja vozlišča, saj se za liste izračuna  $globinaNiza(v)$ , kot  $sd(v) = n - SA[rang_{01}(\tau, v)] + 1$ . Torej operacija potrebuje  $O(t_{SA})$  časa, da se izvrši, saj je v obeh primerih potreben dostop do priponskega polja, za katerega je potrebno  $O(t_{SA})$  časa.

Operacija  $lca(v, w)$ , ki vrne najgloblje vozlišče  $u$ , ki je hkrati predhodnik od  $v$  in od  $w$ , ima enako časovno zahtevnost v obeh primerih. Pri tem priponsko drevo potrebuje dodatno podatkovno strukturo, ki je izgrajena v času  $O(n)$ . V priponskem drevesu je lahko operacija  $lca(v, w)$  naivno implementirana s pomočjo primerjanja vozlišč, na podoben način kot je implementirana v kompaktni predstavitvi LOUDS, kar je prikazano v Algoritmu 1. Pri tem pa se primerjata globina vozlišča in kazalca na vozlišče namesto položaj vozlišča v zaporedju. Operacija je lahko izboljšana na konstantni čas s pomočjo algoritma, ki sta ga zapisala Harel in Tarjan [19]. Dodatna podatkovna struktura  $L$  (ime je enako kot v kompaktnem priponskem drevesu) potrebuje  $O(n)$  dodatnega prostora ter mora omogočati  $RMQ$  operacije. Implementirana je na podoben način kot v kompaktnem priponskem drevesu. Torej se operacija  $lca(v, w)$  tako v priponskem drevesu kot tudi v kompaktnem priponskem drevesu zapiše kot

$$lca(v, w) = rmq(L, v, w) = u,$$

pri čemer so  $v, w$  in  $u$ , v priponskem drevesu, zaporedno števila obiskanih istoimenskih vozlišč v sprehodu po priponskem drevesu. Zato mora biti  $u$  pretvorjen nazaj v kazalec na vozlišče. V kompaktnem priponskem drevesu pa sprehod in pretvorba v kazalec nista potrebna, saj je  $u$  predstavljen kot položaj uklepaja v zaporedju uravnoveženih oklepajev  $\tau$ . V obeh primerih z uporabo dodatne podatkovne strukture  $L$  je možno izračunati  $lca(v, w)$  v konstantnem času.

Zadnja predstavljena operacija pa je operacija  $sl(v)$ , ki vrne vozišče  $w$ , na katerega kaže priponska povezava iz vozlišča  $v$ . V priponskem drevesu so priponske povezave in posledično operacija  $sl(v)$  implementirane, tako da vozlišče  $v$  hrani kazalec na vozlišče  $w$ , kar predstavlja priponsko povezavo. Torej je operacija  $sl(v)$  implementirana s sledečo psevdokodo `v.priponskaPovezava`. V kompaktnih priponskih drevesih pa je potrebno naračunati priponsko povezavo vozlišča  $v$ . Torej se lahko operacijo  $sl(v)$  implementira kot:

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

Ker operacije  $lca(v, w)$ ,  $izbira_{01}(\tau, x)$ ,  $rang_{01}(\tau, x)$  in  $zapri(\tau, x)$  potrebujejo konstantni čas, da se izvršijo, je časovna zahtevnost  $O(t_\Psi)$ . Za izračunati vrednost  $\Psi(\cdot)$  je potrebno  $O(t_\Psi)$  časa, kar je odvisno od implementacije kompaktne priponskega polja. V predhodno predstavljeni implementaciji kompaktne priponskega polja je  $t_\Psi = O(1)$  in se zato v obeh primerih operacija  $sl(v)$  izvrši v konstantnem času.

Naslednji razdelek v Tabeli 4 predstavlja osnovne poizvedbe, ki se izvajajo nad priponskimi drevesi. Najbolj preprosta poizvedba med njimi je *prisotnost(vzorec)*, ki preveri, če je vzorec *vzorec* dolžine  $m$  prisoten v besedilu  $T$ . To poizvedbo se lahko izvede direktno nad besedilom z uporabo *KMP* algoritma, ki potrebuje  $O(n + m)$  časa,

kar je enako kot čas potreben za izgraditi priponsko drevo,  $O(n)$ , in preveriti prisotnost vzorca v drevesu,  $O(m)$ . Ko želimo preveriti prisotnost večjega števila vzorcev v besedilu, uporabimo priponsko drevo. Prisotnost vzorca v besedilu s priponskim drevesom se preveri s sprehodom iz korena proti listom drevesa. Pri tem se preveri, ali se nizi na povezavah ujema z iskanim vzorcem. V vsakem novem vozlišču je potrebno najti otroka, ki se začne z naslednjim znakom v vzorcu, za kar je potrebno  $O(\log |\Sigma|) = O(1)$ , saj je velikost abecede konstantna. Potrebno je še preveriti, ali je celoten vzorec *vzorec* prisoten v priponskem drevesu, kar ima časovno zahtevnost  $O(m \log |\Sigma|) = O(m)$ .

---

**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  $i = m..1$ 
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 return  $[s, e]$ 
```

---

V kompaktnem priponskem drevesu se prisotnost vzorca išče s pomočjo vzratnega iskanja (angl. *Backward Search*) vzorca v priponskem polju  $SA$ . Vzratno iskanje za predhodno predstavljeno kompaktno priponsko polje, ki je prikazano s Algoritmom 6, potrebuje  $O(mt_\Psi)$  časa, da se izvrši. Za drugačno implementacijo kompaktne priponskega polja, se nadomesti vrstico 6 v Algoritmu 6 z binarnim iskanjem nad  $\Psi_c$  in zato je potrebno  $O(m \log nt_\Psi)$  časa. Nato pa je potrebo preveriti, ali je  $[s, e] \neq [-1, -1]$ , za kar je potrebno konstantno časa. Torej je potrebno  $O(mt_\Psi)$  časa za preveriti prisotnost vzorca ali  $O(m \log nt_\Psi)$  z uporabo binarnega iskanja. V primeru, da se uporabi opisano kompaktno priponsko polje, pa se poizvedba izvrši v času  $O(m)$ .

Naslednja poizvedba je *številoPonovitev(vzorec)*, ki vrne število pojavov vzorca v besedilu, kar je ekvivalentno številu pripon v priponskem drevesu, ki se začnejo z vzorcem *vzorec*. V priponskem drevesu je potrebno najti vozlišče  $v$ , za katerega velja  $besedilo(v)[1, m] = vzorec$ . Po tem takem je število ponovitev vzorca enako številu listov v poddrevesu s korenem v vozlišču  $v$ . Štetje vseh listov zahteva  $O(n)$  časa, za iskanje vozlišča  $v$  pa je potrebno  $O(m)$  časa, saj iskanje poteka na isti način, kot v poizvedbi *prisotnost(vzorec)*. V kompaktnem priponskem drevesu, pa je operacija ponovno implementirana s pomočjo vzratnega iskanja. Operacija *številoPonovitev(vzorec)* je



implementiran kot razlika  $e - s$ , kjer  $s$  predstavlja prvo pripono, ki se začne z vzorcem *vzorec*, in  $e$  je zadnja tako pripona, torej je razlika  $s - e$  število pripon, ki se začnejo z vzorcem *vzorec*. Torej operacija ponovno potrebuje  $O(mt_\Psi)$  oziroma  $O(mt_\Psi \log n)$  časa, da se izvrši.

Zadnja predstavljena poizvedba pa je *seznamPojavov(vzorec)*, ki vrne vsa začetna mesta pojavov vzorca *vzorec* v besedilu  $T$ . To je ekvivalentno seznamu vseh pripon besedila  $T$ , ki se začnejo z vzorcem *vzorec*. V priponskem drevesu je to implementirano na podoben način, kot poizvedba *številoPonovitev(vzorec)*, pri čemer namesto štetja listov v poddrevesu s korenem v vozlišču  $v$ , se v seznam pripon dodaja vse pripone, ki so predstavljene kot listi v poddrevesu s korenem v vozlišču  $v$ . V kompaktnem priponskem drevesu pa je poizvedba ponovno implementirana s pomočjo vzvratnega iskanja. Z vzvratnim iskanjem se naračuna interval v priponskem polju  $SA[s, e]$ . Za pridobitev položajev ponovitev vzorca v besedilu, je potrebno ustvariti seznam  $[SA[s], SA[s+1], \dots, SA[e]]$ , kar zahteva še dodatnih  $e - s$  korakov, pri čemer vsak korak potrebuje  $O(t_{SA})$  časa za se izvršiti. Torej poizvedba *seznamPojavov(vzorec)* potrebuje  $O(mt_\Psi + t_{SA})$  oziroma  $O(mt_\Psi \log n + t_{SA})$  časa. Z predhodno predstavljeno implementacijo kompaktnega priponskega drevesa pa je potrebno  $O(m + \log n)$  časa.

Iz Tabele 5 je razvidno, da sta poizvedbi *seznamPojavov(vzorec)* in *številoPonovitev(vzorec)* hitrejši v kompaktnih priponskih drevesih, natanko tedaj ko je velikost vzorca  $m$  bistveno manjša od dolžine besedila  $n$ . Poizvedba *prisotnost(vzorec)* ostane enako hitra za obe implementaciji kompaktnega priponskega polja tudi za velikosti vzorca  $m = O(n)$ . Pri tem pa ostane velikost abecede  $\Sigma$  ne spremenjena, torej je tudi  $O(\log |\Sigma|) = O(1)$ .

Iz implementacij poizvedb nad kompaktnim priponskim drevesom se lahko vidi, da so vse tri poizvedbe implementirane zgolj s pomočjo kompaktnega priponskega polja. Iz tega se lahko sklepa, da sta topologija drevesa  $\tau$  in  $LCP$  polje odvečni podatkovni strukturi. To je res zgolj za osnovne poizvedbe nad besedilom  $T$ , ki so lahko izvršene zgolj s priponskim polje v enakem času. Poizvedbe, kot so najdaljši ponavljajoči podniz, najdaljši palindrom, ki je implementirana s pomočjo priponskega drevesa konkatencije obrata  $T'_z$  besedila  $T_z$ ,  $T = T_z \# T'_z \$$ , in najdaljši skupni niz besedila  $T_1$  in  $T_2$ , ki je implementirana s pomočjo priponskega drevesa konkatencije obeh besedil  $T = T_1 \# T_2 \$$ . Na primer poizvedbe najdaljši ponavljajoči podniz je podniz  $T[SA[i], SA[i] + globinaNiza(v)]$ , pri čemer  $i$  je 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 *globinaNiza(v)* ni potrebno naračunati, saj je enaka  $LCP[i]$ , torej se še vedno izvede  $O(nt_{SA})$  časa, pri tem pa ni uporabljena topologija drevesa. V primeru, da želimo najti drugi najdaljši ponavljajoč se podniz  $T[SA[j], SA[j] + globinaNiza(u)]$ , pri čemer je  $j$  skrajno levi list poddrevesa s korenem v notranjem vozlišču  $u = sl(v)$ . Za izračunati le tega pa je potrebo  $O(nt_{SA} + t_\Psi)$  časa ter se uporabi

vse tri podatkovne strukture kompaktnega priponskega drevesa [3, 5, 6].

V zadnjem razdelku Tabele 4 pa sta prikazani še dve primerjavi. Prva primerjava je časovna zahtevnost izgradnje priponskega drevesa ali kompaktnega priponskega drevesa iz besedila  $T$  dolžine  $n$ . Izrek 3.7 trdi, da je možno priponsko drevo izgraditi s pomočjo Ukkonenovega algoritma v času  $O(n)$ . Kompaktno priponsko drevo pa se po Izreku 5.9 lahko izgradi v času  $O(n \log n)$ . Čeprav bi se lahko kompaktno priponsko drevo izgradilo v času  $O(n)$  s pomočjo priponskega drevesa. Tako izdelan algoritem ne ohranja kompaktne prostorske zahtevnosti skozi celotno izgradnjo, za razliko od predstavljenega algoritma. Algoritem za izgradnjo kompaktnega priponskega drevesa je za  $O(\log n)$ -krat počasnejši od algoritma za izgradnjo priponskega drevesa, vendar lahko izgradi priponsko drevo za večja vhodna besedila za razliko od Ukkonenovega algoritma za isto količino spomina.

Iz primerjave algoritmov je razvidno, da se lahko za isto količino spomina izgradi kompaktno priponsko drevo za daljše besedilo kot za priponsko drevo. Priponsko drevo potrebuje  $O(n)$  povezav. Čeprav se  $O(n)$  povezav sliši manj kot  $|CSA| + 6n + o(n)$  bitov za kompaktno priponsko drevo, ima v resnici vsaka povezav velikost  $O(\log n)$  bitov, kar pomeni, da je prostorska zahtevnost priponskega drevesa  $O(n \log n)$  bitov. Kompaktno priponsko drevo pa potrebuje  $|CSA| + 6n + o(n)$  bitov, pri čemer so velikosti kompaktnega priponskega polja  $|CSA|$  zapisane v tabeli 5. V primeru, da vzamemo za primerjavo prostorsko manj učinkovito kompaktno priponsko polje [16], potem je prostorska zahtevnost kompaktnega priponskega drevesa  $n \log |\Sigma| + 6n + o(n)$  bitov, kar je  $O(n)$  bitov, če ostane abeceda konstanta skozi celoten čas obstoja drevesa. Potemtakem je prostorska zahtevnost priponskega drevesa  $O(\log n)$ -krat večja od prostorske zahtevnosti kompaktnega priponskega drevesa.

Ampak so to le teoretične primerjave časovne zahtevnosti operacij, poizvedb in izgradnje ter prostorske zahtevnosti priponskih dreves, zato je potrebno te primerjave potrditi z empirično primerjavo. Z empirično evalvacijo se želi ugotoviti vpliv delovnega spomina, v katerega je zaradi lažje analize štet tudi **swap** razdelek zunanega spomina, na časovne zahtevnosti različnih implementacij priponskega drevesa ter prostorsko zahtevnost različnih implementacij priponskega drevesa v vsakdanji uporabi.

## 6.2 OPIS METODE EMPIRIČNE PRIMERJAVE

V tem poglavju je opisana empirična primerjava časovnih zahtevnosti različnih operacij, poizvedb in izgradnje ter prostorske zahtevnosti različnih implementacij priponskega drevesa. Priponsko drevo se uporablja za iskanje vzorcev v besedilu  $T$ , torej se je smiselno osredotočiti zgolj na primerjavo časovnih zahtevnosti za poizvedbe ter za izgradnjo priponskega drevesa. Primerjava posamičnih operacij priponskega drevesa

nima smisla, saj se le te uporabljajo pri implementaciji poizvedb in izgradnji.

Pri tem se bomo osredotočili zgolj na osnovne poizvedbe. Kot najbolj preprosta osnovna operacija bo izdelana primerjava nad poizvedbo  $prisotnos(vzorec)$ . Ta poizvedba je bila izbrana, saj je pogosto vprašanje v biologiji prisotnost genov (vzorcev) v DNK sekvenci, ne pa natančen položaj tega gena ali števila ponovitev vzorca. Čas izvajanja poizvedbe bo izmerjen kot razlika v času takoj pred začetkom izvajanja poizvedbe ter takoj po zaključku izvajanja poizvedbe. Poizvedba bo izmerjena na vzorcih dolžine 5 znakov, 50 znakov, 500 znakov in  $\log n$  znakov, kjer je  $n$  dolžina znaka.

Na podoben način kot poizvedba  $prisotnos(vzorec)$  bo izmerjen tudi čas izgradnje priponskega drevesa. Le ta bo izmerjen s pomočjo razlike v uri med časom ure takoj pred izgradnjo ter v času ure takoj po izgradnji priponskega drevesa.

Izmerjeni časi bodo shranjeni v vektorju potrebnih časov  $T_{i,v}$ , pri čemer  $i$  predstavlja dolžino vhodnega besedila in  $v$  predstavlja tip priponskega drevesa (priponsko drevo z vrednostjo  $v = ST$  ali kompaktno priponsko drevo z vrednostjo  $v = CST$ ). Vektor bo sestavljen na sledeči način

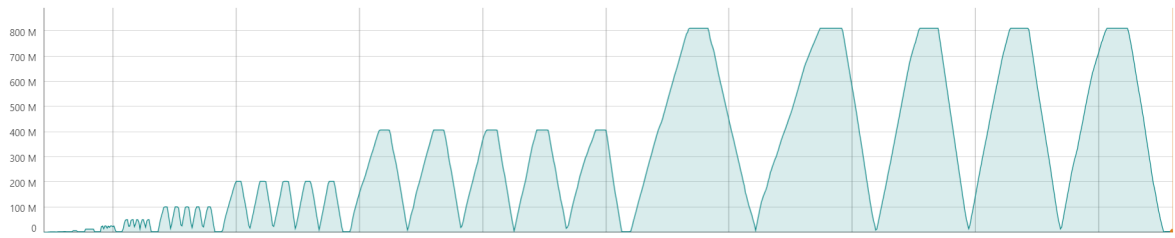
$$T_{i,v} = (t_{izg}, t_5, t_{50}, t_{500}, t_{\log n}),$$

pri čemer so izmerjeni časi za poizvedbe izmerjeni v nanosekundah, čas izgradnje  $t_{izg}$  pa v milisekundah.

Na podoben način bo shranjena tudi velikost, ki jo zasede priponsko drevo v delovnem spominu. Izmerjen prostor bo poleg prostora priponskega drevesa vseboval tudi besedilo  $B_0$ , katerega podniz  $B_0[1, i]$  bo uporabljen za izgradnjo. Ker bodo vse meritve vsebovale velikost besedila  $B_0$ , le to ne bo vplivalo na primerjavo podatkov. Izmerjen prostor bo shranjen v vektorju  $S_{i,v}$ , pri čemer  $i$  predstavlja dolžino vhodnega besedila in  $v$  predstavlja tip priponskega drevesa (priponsko drevo z vrednostjo  $v = ST$  ali kompaktno priponsko drevo z vrednostjo  $v = CST$ ). V tem primeru bo vektor  $S_{i,v}$  vseboval zgolj vrednost  $s_{max}$ , ki je največja dosežena velikost priponskega drevesa med izvajanjem izgradnje in poizvedbe. Največji izmerjeni zasedeni prostor na delovnem spominu bo izmerjen z uporabo pomnilniškega profilerja (angl. *memory profiler*).

Uporabljeni bo Bytehound [20] pomnilniški profiler, saj omogoča grafični prikaz porabe spomina v času izvajanja programa. Primer prikaza porabe delovnega spomina v času izvajanja testa je prikazan na Sliki 14. Na sliki se jasno vidi, da večanje priponskega drevesa povzroči rast zasedenega prostor na delovnem spominu v času izgradnje in poizvedbe. Na sliki se lahko opazi, da izgradnja in poizvedbe v kompaktnem priponskem drevesu potrebujejo krepko manj časa in prostora, zato izgleda, kot da se graf konča malo pred koncem vodoravne osi. Razlog za to je izgradnja in poizvedba nad kompaktnim priponskim drevesom, ki potrebuje manj pomnilnika in posledično manj časa za dodeliti in sprostiti delovni spomin.

Za zmanjšati vpliv ostalih procesov na računalniku, se je vsak test ponovil 5-krat.



Slika 14: Zasedenost spomina testiranjem različnih implementacij priponskega drevesa skozi celotno izvajanje testa.

To je tudi vidno na Sliki 14, zato ima zadnje izgrajeno priponsko drevo 5 vrhov in vsak vrh predstavlja eno ponovitev testiranja. S pomočjo profilerja je bila tudi izmerjena velikost originalnega besedila  $B_0$ , ki je prikazan v zadnjem stolpcu Tabele 6, saj se velikost razlikuje glede na vhodno besedilo.

Tabela 6: Primerjava besedil, ki bodo uporabljena za primerjavo različnih implementacij priponskih dreves

Ime testnega besedila	Število znakov	Velikost abecede	Velikost na disku [MB]
Ivan Cankar, Na klancu [10]	317803	52	25,851
DNK sekvenca [9]	52428800	4	26,767

S tem načinom testiranja se želita ugotoviti dve stvari. Najprej se želi ugotoviti vpliv velikosti vhodnega besedila na čas izgradnje in poizvedb nad priponskim drevesom ter velikost spomina, ki ga zasede priponsko drevo. To bo izmerjeno z izgradnjo različnih dreves ter s poizvedbami nad njimi. Prvo drevo bo izgrajeno nad besedilom dolžine 500 znakov. Vsako naslednje priponsko drevo bo izgrajeno nad besedilom, ki je dvakrat daljše od predhodnega. Zadnje izgrajeno priponsko drevo bo imelo dolžino besedila, za katerega je bilo izgrajeno, manjšo od  $|B_0| - 500$  znakov ali  $|B_0| - \log(500 \cdot 2^{i-1})$  za besedila daljša od  $2^{500}$  znakov. V besedilu mora bit vedno na voljo dovolj znakov, da se bodo lahko iz njih naredili vzorci vseh testiranih velikosti. Besedilo, ki bo uporabljeno za izgradnjo  $i$ -tega priponskega drevesa, bo podniz  $B_0[1, 500 \cdot 2^{i-1}]$ .

Druga stvar, ki se želi ugotoviti, je vpliv uporabe **swap** razdelka za namene delovnega spomina ter vpliv velikosti abecede vhodnega besedila in vzorca na iskanje vzorca v priponskem drevesu. Zato bo primerjava narejena na dveh besedilih, ki imata različno vhodno abecedo, in sicer prvo vhodno besedilo je kratki roman Ivana Cankarja, Na klancu [10], ki uporablja slovensko abecedo, ter daljša DNK sekvenca [9], ki pa uporablja abecedo  $\Sigma = \{A, C, T, G\}$ . Pri tem obe abecedi ne vsebujeta znaka, ki predstavlja konec besedila, \$, saj bo le ta dodan besedilu pred začetkom izgradnje. Več podatkov o vhodnih besedilih je prikazanih na Tabeli 6.

### 6.2.1 Pred obdelava besedil

Kot je razvidno v drugem stolpcu Tabele 6 je potrebno najkrajše besedilo podaljšati. Ker je malo verjetno, da se vhodno besedilo ponovi  $k$ -krat, dokler ne doseže primerne velikosti, predvsem v naravnem jeziku, bo uporabljena bolj napredna metoda podaljševanja besedila. Metoda vzame manjše dele besedila dolžine  $5i$  ter jih konkatenera na koncu besedila. Tako dobljeno besedilo je bolj verjetno, saj je večja verjetnost, da se manjši deli besedila ponovijo za razliko od celotnega besedila. Predlagana metoda podaljševanja besedila je prikazana z Algoritmom 7. Pri tem metoda predpostavi, da je besedilo dolgo vsaj  $6i$ , kar je 3000 znakov dolgo. Ta metoda je primerna za podaljševanje daljših besedil, sicer pa je možno metodi spremeniti parameter  $i$  in tako prilagoditi metodo drugim besedilom.

---

**Algoritem 7:** Metoda podaljševanja vhodnega besedila
 

---

**Vhod:** Vhodno besedilo  $B$ , želena velikost  $s_{max}$

**Izhod:** Besedilo  $B_0$

```

1  $B_0 = B$ 
2  $i = 500$ 
3 while  $|B_0| < s_{max}$  do
4    $B_0 = B_0 + B[i, 6i]$ 
5    $i = i + 500$ 
6   while  $6i > |B|$  do
7      $i = i/4$ 
8 vrni  $B_0[1, s_{max}]$ 
```

---

Predlagana metoda podaljša besedilo na velikost  $s_{max}$ . Ta velikost je lahko dolžina najdaljšega besedila ali pa je poljubna vrednost, bodisi manjša od največjega besedila bodisi večja. Če je besedilo daljše od velikosti  $s_{max} < |B|$ , bo predlagana metoda skrajšala velikost besedila na  $|B_0| = s_{max}$ .

### 6.2.2 Iskanje vzorcev

Po izgradnji priponskega drevesa bo le to uporabljeno za iskanje vzorcev v besedilu. Kot je bilo predhodno omenjeno, se bo pregledovala zgolj prisotnost vzorca v besedilu. Velikosti vzorcev, ki bodo iskani v besedilu so 5, 50 in 500 znakov ter  $\lfloor \log(500 \cdot 2^{i-1}) \rfloor$  znakov, pri čemer je  $i$  zaporedna številka testa velikosti priponskega drevesa. Vzorec dolžine  $x$  ( $x$  je ena od predhodno naštetih velikosti vzorca) je pridobljen kot  $B_0[500 \cdot 2^{i-1} + 1, 500 \cdot 2^{i-1} + 1 + x]$ .

Vzorci, ki so izdelani na tak način, zagotavljajo, da z visoko verjetnostjo niso prisotni v besedilu ter posledično niti v priponskem drevesu. To naredi test bolj realističen, saj

ko vemo, da je vzorec prisoten v besedilu, nima smisla preverjati pristnosti vzorca. Če pa je besedilo  $B_0$  podaljšano, na kakršen koli način, je prisotnost vzorca v besedilu večja.

## 6.3 REZULTATI EMPIRIČNE PRIMERJAVE

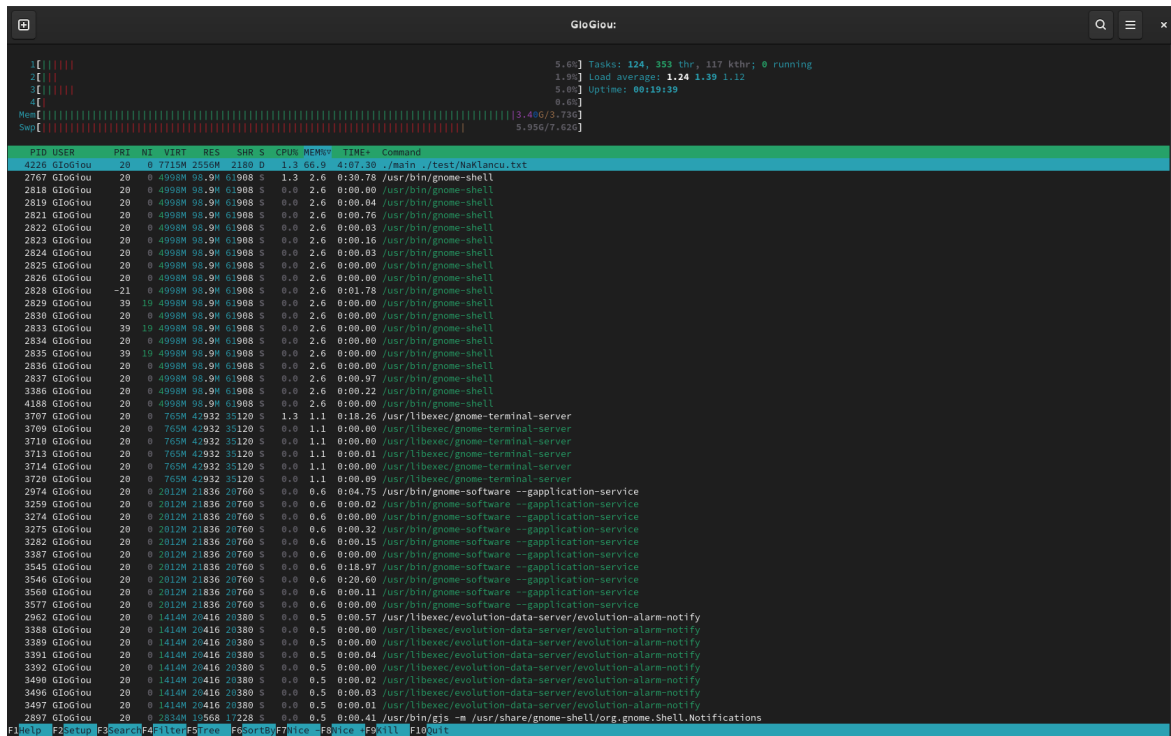
Sedaj predstavljena metoda empirične primerjave, se lahko uporabi za empirično primerjavo med priponskimi drevesi in kompaktnimi priponskimi drevesi. Primerjava je bila izdelana s programskim jezikom C++<sup>1</sup>. Priponska drevesa so bila izgrajena z Ukkonenovim algoritmom implementiran v C++ knjižnici [12]. Kompaktna priponska drevesa so bila izgrajena s predhodno predstavljeno metodo. Uporabljena je bila implementacija kompaktnih priponskih dreves iz C++ knjižnice [11].

Evalvacija je bila izvedena na računalniku s procesorjem Intel Core i3 5005U z dvema jedroma in štirimi nitmi ter s taktom 1,9 GHz. Računalnik ima na razpolago 4 GB delovnega spomina, od katerega je ob zagonu računalnika zasedenih 1,76 GB, ter ima še dodatnih 8 GB **Swap** razdelka na trdem disku. Operacijski sistem računalnika je Fedora 41, ki uporablja Linux kernel 6.11.10-300.

Ob testiranju izgradnje večjih priponskih dreves je bilo opaženo, da operacijski sistem ubije (angl. *kill*) proces, ki izvaja testiranje, zaradi nevarnosti o preseganja velikosti delovnega spomina. To se je zgodilo pri izgradnji priponskega drevesa velikosti 4000000 znakov. Pri tem se je tudi opazilo, da pri priponskem drevesu velikosti 2048000 znakov ni mogoče izgraditi drevesa do konca, saj računalnik ne uspe premakniti strani iz delavnega pomnilnika na **Swap** razdelek in obratno. To naredi računalnik neodziven in proces je v neprekinjenem spanju (angl. *Uninterruptible sleep* ali stanje D), pri tem računalnik doseže 6 GB zasedenega prostora na **Swap** razdelku. Na Sliki 15 je prikazan upravljalnik opravil Htop v času izgradnje priponskega drevesa za besedilo velikosti 2048000 znakov. Proces v modri vrstici predstavlja program za testiranje izgradnje besedil. V stolpcu označenim S (Stanje ali angl. *Status*) je vidno, da je je stanje procesa označeno kot D, ker je proces v neprekinjenem spanju. Po več kot 5 minutah od začetka izgradnje prvega priponskega drevesa sem se odločil, da se proces ubije. Posledično se je znižala velikost besedil za izgradnjo zadnjega priponskega drevesa na 1024000 znakov.

---

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



Slika 15: Posnetek zaslona upravljalnika opravil Htop med izgradnjo priponskega drevesa za besedilo dolžine 2048000 znakov.

Pri tem se je tudi omejila velikost besedila  $B_0$  na  $S_{max} = 25000000$  znakov. To zagotavlja dovolj znakov za izgradnjo vhodnih besedil ter za izgradnjo besedil, ki bodo predstavljali vzorce. Ker je DNK sekvenca [9] daljša od velikosti  $S_{max}$ , je bila le ta odrezana na velikost  $S_{max}$  in sicer bo uporabljeni le prvih 25000000 znakov. Za razliko od DNK sekvence, je besedilo Na klanecu [10] moralo biti podaljšano na  $S_{max} = 25000000$  znakov. To je bilo storjeno z implementacijo Algoritma 7 v programskem jeziku C++. Pri tem ni bilo potrebno implementirati preverjanja, ali šestkratnik števca  $i$  presega velikost besedila  $B$ . Torej tega nisem storil.

Čas potreben za izgradnjo priponskega drevesa in izvršitev poizvedb je bil izmerjen s pomočjo razlike v uri pred in po izvršitvi testa, kot je to bilo predhodno opisano. Meritev je bila implementirana s funkcijo `high_resolution_clock::now()`, ki je del standardne knjižnice programskega jezika C++ in vrne natančen čas trenutka, v katerem je izvedena. S tako izmerjenim časom pred začetkom in takoj po koncu izvajanja se lahko naračuna razlika s funkcijo `duration_cast<milliseconds>(stop - start).count()`, kjer `stop` predstavlja čas konca izvajanja in `start` pa predstavlja čas začetka izvajanja operacije. Funkcija vrne v tem primeru razliko med tema dvema trenutkoma v milisekundah, ki pa se lahko po potrebi zamenja v druge časovne enote. V primeru poizvedb je bil čas iskanja vzorca izmerjen v nanosekundah in je bil implementiran s pomočjo funkcije `duration_cast<nanoseconds>(stop - start).count()`.

Vektorja rezultatov testiranja  $T_{i,v}$  in  $S_{i,v}$  sta bila zapisana v CSV (angl. *Comma-*

*separated values*) datoteko za lažjo nadaljnjo obdelavo rezultatov. Vsaka vrstica datoteke predstavlja rezultat enega testiranja. Poleg vektorjev  $T_{i,v}$  in  $S_{i,v}$ , je v vsaki vrstici zapisana velikost začetnega besedila  $i$  ter vrsta priponskega drevesa  $v$ . Ker se med izvajanjem programa ne beleži velikost priponskega drevesa, se zato zapiše začasna vrednost.

Predhodno predstavljena implementacija je bila izdelana v datoteki `main.cpp`. Koda je bila prevedena iz programskega jezika C++ v izvršljivo datoteko z uporabo prevajalnika (angl. *compiler*) GCC 14.2.1. Program je bil preveden s sledečim ukazom:

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

Pri prevajanju programa je uporabljenih nekaj zastavic, ki določajo vrednosti parametrov prevajalnika. Večina zastavic je vezana na uvažanje knjižnic v prevajanje, in sicer `-I ./include` nastavi pot do zaglavnih datotek (angl. *header files*), `-L ./lib` nastavi pot do strojne kode knjižnice ter zastavice `-lsdsl`, `-ldivsufsort`, `-ldivsufsort64` in `-lsuffix` uvozijo potrebne knjižnice za izgradnjo izvršljive datoteke. Zastavica `-O3` določa nivo optimizacije izvršljive datoteke, na nivo 3, ki je bil izbran, saj je uporabljen za prevajanje prve implementacije kompaktne priponskega drevesa [3]. Pri izgradnji programa je bila uporabljena standardna različica C++11 določena z zastavico `-std=c++11`. Razlog za izbiro starejše različice programskega jezika je knjižnica SDSL [11] (uporabljena kot implementacija kompaktnih priponskih dreves), ki je implementirana za to različico.

Ker tako prevedena datoteka ne omogoča neposrednega nadzora velikosti podatkovnih struktur med izvajanjem, bo za ta namen uporabljen profiler spomina. Zato je potrebo pred začetkom testiranja priponskih dreves zagnati tudi profiler, kar je storjeno z ukazom:

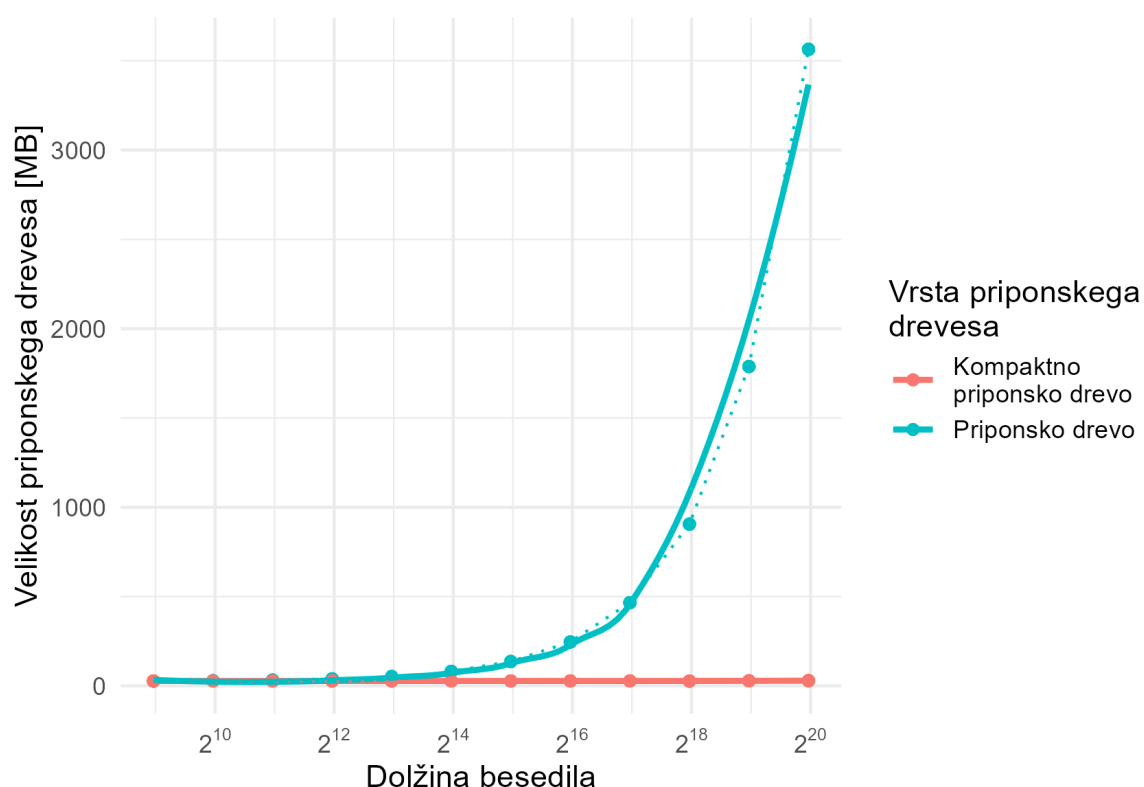
```
export MEMORY_PROFILER_LOG=info
LD_PRELOAD=~/.bytehound/libbytehound.so ./main //
./test/NaKlancu.txt
```

Prva vrstica ukaza določa stopnjo profiliranja, ki ima v tem primeru vrednost `info`, ter jo shrani v sistemsko spremenljivko `MEMORY_PROFILER_LOG`. Naslednja vrstica požene program na datoteki s testnimi podatki, ki so podani kot prvi argument (ime datoteke). Zgornji ukaz prikaže primer za vhodno besedilo Na klancu. S spremenljivko `LD_PRELOAD` je določen deljeni objekt (angl. *shared object*), ki je izvršen pred začetkom programa. V primeru testiranja priponskih dreves je deljeni objekt profiler, kar mu omogoča dostop do programa in lažje beleženje zasedenega delovnega spomina.



Prva izdelana primerjava priponskega drevesa in kompaktnega priponskega drevesa je izdelana nad 50 MB dolgo DNK sekvenco [9]. Kot vsa ostala testna besedila, je tudi DNK sekvenca shranjena v mapi `./test/` pod imenom `DNA.50MB`. Izbrana je bila, saj so jo uporabili kot testno besedilo v implementaciji od Välimäki idr. [3]. Sekvenca je zlepek različnih DNK sekvenc.

Kot je bilo predstavljeno v Tabeli 6 je besedilo sestavljeno iz  $4 + 1$  znakov. Izmerjen je bil tudi potreben prostor za shranjevanje vhodnega besedila, ki zasede 26,767 MB delovnega spomina. Velikost zasedenega prostora na delovnem spominu z vhodnim besedilom je potrebno odšteti od velikosti, ki jo potrebujejo implementirana priponska drevesa.

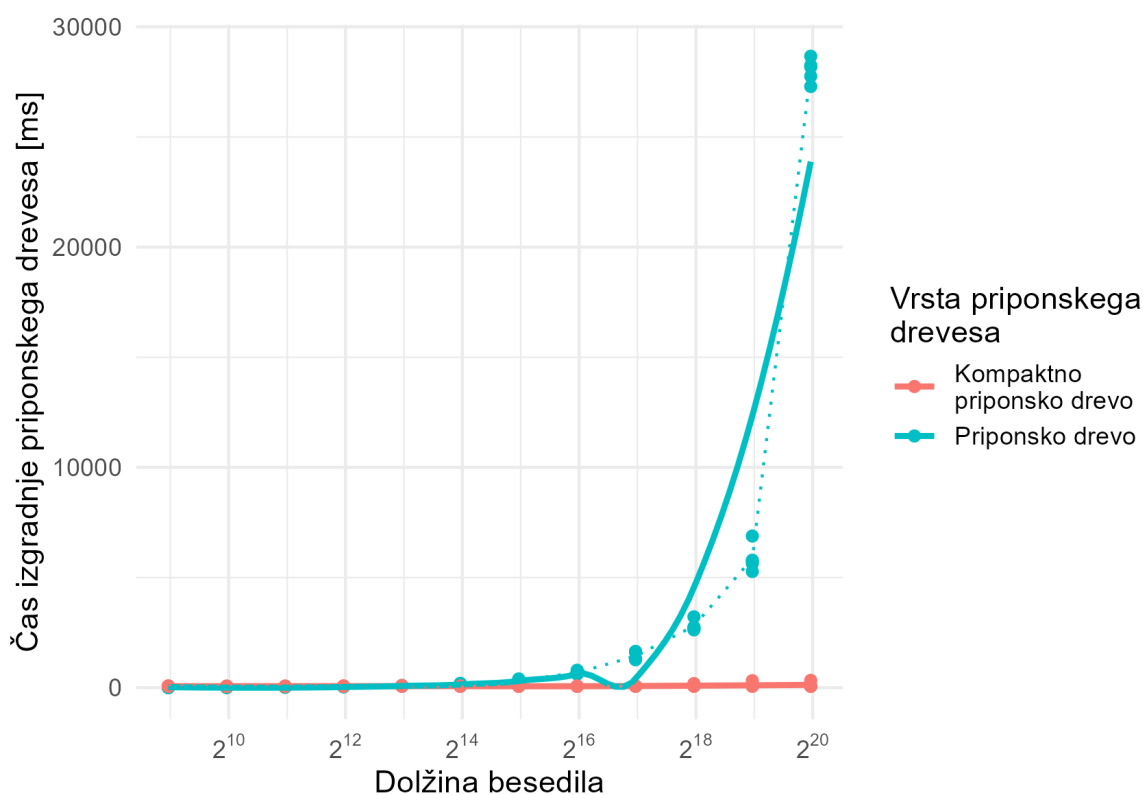


Slika 16: Graf velikosti priponskega drevesa izgrajenih iz besedil različne velikosti. Vhodno besedilo je DNK sekvenca.

Na Sliki 16 je prikazana potrebna količina delovnega spomina za izgradnjo priponskega drevesa, ki je označena z modro, ter kompaktnega priponskega drevesa, ki pa je označena z rdečo barvo. Na sliki je uporabljena, na vodoravni osi, logaritmčna skala za velikost besedila, saj je vsako naslednje testirano besedilo dvakrat večje od predhodnega. Iz slike se lahko jasno vidi, da za besedila do dolžine 8000 znakov obe drevesi zasedeta približno enako količino prostora na delovnem spominu, in sicer priponsko drevo zasede 39,8 MB in kompaktno priponsko drevo zasede 27,4 MB. Vse nadaljnje razlike v velikosti med priponskim drevesom in kompaktnim priponskim drevesom so

izrazite. Čeprav se zdi, da je velikost kompaktne priponskega drevesa konstanta, se ta v času izvajanja testiranja dvigne iz približno 27,4 MB na približno 29 MB.

Na Sliki 16 se lahko tudi opazi, da velikost priponskega drevesa ne presega 4 GB, kar je velikost notranjega spomina. Pri tem pa je potrebno upoštevati dejstvo, da operacijski sistem potrebuje 1,76 GB delovnega spomina takoj po zagonu računalnika. To pomeni, da čeprav velikost priponskega drevesa ne presega velikosti delovnega pomnilnika, mora biti le ta shranjen v **Swap** razdelku, saj mora računalnik v vsakem trenutku zagotoviti dovolj prostora na delovnem pomnilniku za operacijski sistem, torej ima program na razpolago največ 2,24 GB delovnega pomnilnika. Zadnje testirano priponsko drevo potrebuje 3,56 GB dolgovnega pomnilnika, zato mora biti vsaj 1,22 GB drevesa shranjenega v na **Swap** razdelku.



Slika 17: Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je DNK sekvenca.

Ker je zadnje izgrajeno priponsko drevo, moralo biti shranjeno tudi na **Swap** razdelku, je potrebno preveriti, na kakšen način to vpliva na čas izgradnje drevesa ter iskanje v njem. Rezultati testiranja časa izgradnje različnih implementacij priponskega drevesa so prikazani na Sliki 17, kjer z modro barvo je prikazan čas potreben za izgradnjo priponskega drevesa ter z rdečo pa čas potreben za izgradnjo kompaktne priponskega drevesa. Vsi časi, ki so prikazani na sliki, so v milisekundah.

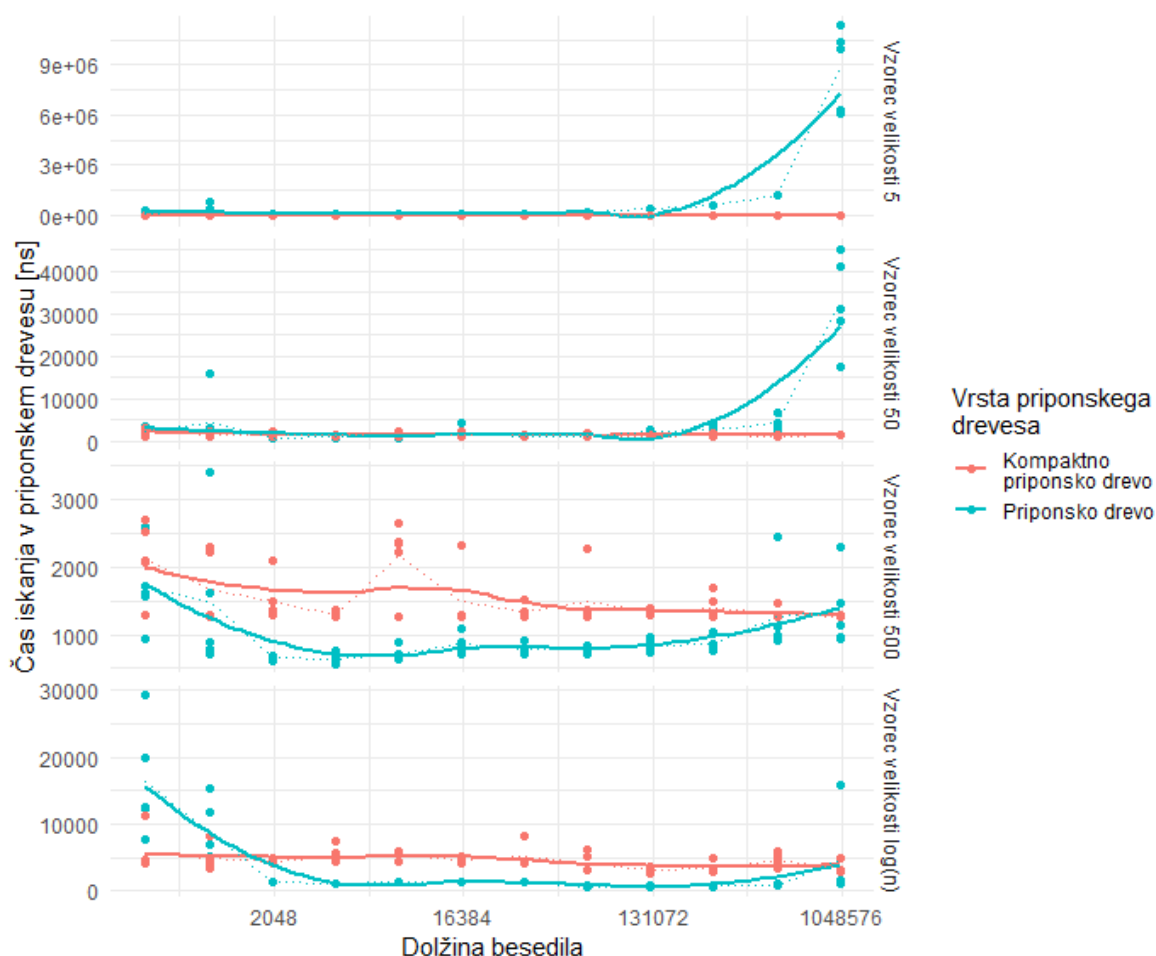
Pri izgradnji največjega priponskega drevesa se lahko opazi na Sliki 17, da je potreben čas višji od pričakovanega. Skok v časovni zahtevnosti se pojavi, pri izgradnji priponskega drevesa, ki je moral biti delno shranjen na **Swap** razdelku. Vsa ostala priponska drevesa so zgrajena dvakrat počasneje kot predhoden priponsko drevo, kar je tudi pričakovano, saj Ukkonenov algoritem izgradi priponsko drevo v času  $O(n)$ . Torej, ker se je velikost besedila podvojila, potemtakem se bo tudi čas izgradnje podvojil. Izjema je zadnje izgrajeno priponsko drevo (dolžina vhodnega besedila je 1024000 znakov), kjer se čas izgradnje poveča za 4,78-krat. Iz tega se lahko sklepa, da uporaba **Swap** razdelka negativno vpliva na čas izgradnje priponskega drevesa.

Iz Slike 17 je tudi razvidno, da čas potreben za izgradnjo kompaktnega priponskega drevesa raste počasneje kot čas izgradnje priponskega drevesa. Čas potreben za izgradnjo kompaktnega priponskega drevesa za besedilo dolžine 500 znakov je 70,8 milisekund, čas za izgradnjo kompaktnega priponskega drevesa za besedilo dolžine 1024000 znakov pa je 121 milisekund. Kompaktno priponsko drevo se zgradi hitreje od priponskega drevesa pri besedilih, ki so daljši od 8000 znakov. Priponsko drevo potrebuje 83,2 milisekunde za izgradnjo, za razliko od kompaktnega priponskega drevesa, ki pa potrebuje 75,2 milisekunde za biti izgrajeno nad besedilom dolžine 8000 znakov.

Nazadnje se lahko še analizira vpliv velikosti priponskega drevesa na potreben čas za izvršiti poizvedb nad njim. Na Sliki 18 so prikazani časi, v nanosekundah, potrebni za iskanje vzorcev, in sicer z modro barvo so predstavljeni časi za poizvedbe v priponskih drevesih, z rdečo pa so predstavljeni časi za poizvedbe v kompaktnem priponskem drevesu. Na sliki so predstavljeni rezultati za vse štiri velikosti vzorcev iskanih v priponskem drevesu: 5 znakov, 50 znakov, 500 znakov in  $\log n$  znakov, pri čemer je  $n$  dolžina besedila.

Na Sliki 18 so prikazani rezultati primerjave iskanja vzorcev dolžine 5 znakov. Časi potrebni za iskanje vzorca so za obe implementaciji priponskih dreves podobni in konstantni. Edina vidna razlika je v času, ki je potrebnem za najdi vzorec v priponskem drevesu dolžine 1024000 znakov. To je posledica uporabe **Swap** razdelka, zato se potrebuje 7,46-krat več čas od ostalih poizvedb s priponskim drevesom. Ostale poizvedbe, za vzorce dolžine 5 znakov, potrebujejo približno 1000 mikrosekund, pri čemer pa priponsko drevo za besedilo dolžine 1024000 znakov pa potrebuje 8,79 milisekund za preveriti prisotnost vzorca. Čas, ki je potreben za poizvedbo v kompaktnih priponskih drevesih, je konstanten in je približno 3,2 mikrosekunde.

Za vzorce dolžine 50 znakov je iz Slike 18 razvidno, da so potrebni časi za iskanje podobni kot pri iskanju vzorcev dolžine 5 znakov. Torej je čas, ki je potreben za iskanje v priponskem drevesu, konstanten in je približno 2,2 mikrosekunde. Ko priponsko drevo preraste delovni spomin in mora bit delno shranjen na **Swap** razdelku se iskalni čas poveča za 7,44-krat. Pri tem iskanje vzorcev dolžine 50 znakov potrebuje približno enako časa kot iskanje vzorcev v kompaktnem priponskem drevesu. V kompaktnem



Slika 18: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je DNK sekvenca.

priponskem drevesu je potrebnih približno 1,5 mikrosekunde za preveriti ali je vzorec prisoten ali ni. Iz tega se lahko sklepa, da za iskanje vzorcev dolžine 50 znakov je bolj učinkovita uporaba priponskih dreves za besedila do 8000 znakov, ko je potrebno tudi izgraditi priponsko drevo. Za besedila daljša od 8000 znakov pa je bolje uporabiti kompaktno priponsko drevo.

Iskanje prisotnosti vzorca dolžine 500 znakov, rezultati so prikazani na Sliki 18 kot predzadnji graf, je bolj učinkovito s priponskimi drevesi. Čeprav je za vse testirane velikosti priponskega drevesa iskanje v priponskem drevesu hitrejše ali primerljivo z iskanjem v kompaktnem priponskem drevesu, se lahko opazi rast v potrebnem času v večjih priponskih drevesih, ki so deloma shranjena v **Swap** razdelku. Čeprav razlika ni tako očitna kot v primeru iskanja vzorca dolžine 5 ali 50 znakov, se lahko vseeno vidi rast iz grafa. Možen razlog za nižjo rast je prisotnost strani (angl. *page*), ki vsebujejo vzorec v delovnem spominu in ne na **Swap** razdelku. Pri tem je vseeno potrebna 1 mikrosekunda za najti vzorec dolžine 500 znakov. Čas iskanja vzorcev v kompaktnih priponskih drevesih še vedno ostane konstantno, pri čemer se potreben čas za iskanje

vzorcev dolžine 500 znakov zniža na 1,5 mikrosekunde. Iz testiranja iskanja vzorcev velikosti 500 znakov v priponskih dresih se lahko sklepa, da če je na razpolago dovolj prostora na delovnem spominu, je boljše uporabiti priponsko drevo za te namene.

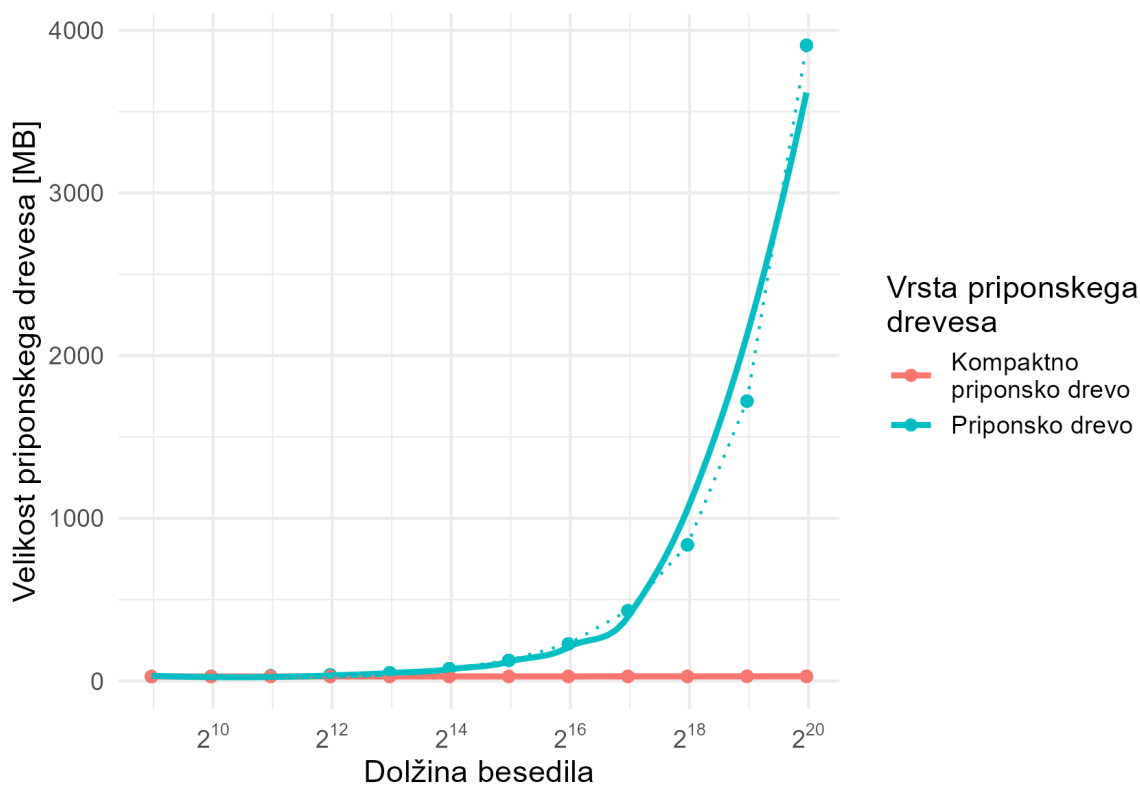
Zadnje testiranje, ki je prikazano na Sliki 18, je iskanje vzorcev dolžine  $O(\log(n))$ , pri čemer je  $n$  velikost besedila, ki je predstavljeno s priponskim drevesom. Velikosti iskanih vzorcev se gibajo od 9 znakov do 20 znakov. Iz grafa je razvidno, da kompaktno priponsko drevo, označeno na grafu z rdečo barvo, potrebuje približno konstanto časa za najti vzorec v priponskem drevesu. Za iskanje vzorca potrebuje približno 4,5 mikrosekunde. Pri tem pa je zgodba bistveno drugačna pri iskanju vzorcev s pomočjo priponskega drevesa, ki je prikazano z modro barvo na sliki. Za vzorce velikosti 9 in 10 znakov je čas iskanja v priponskem drevesu višji kot čas iskanja v kompaktnem priponskem drevesu. V vseh ostalih primerih pa je čas iskanja nižji kot v kompaktnem priponskem drevesu, podobno kot pri iskanju vzorcev dolžine 500 znakov. Iskanje vzorca dolžine 11 znakov ali več potrebuje približno 1 mikrosekundo, da se izvrši. Pri tem se lahko tudi opazi, da je čas iskanja v zadnjem drevesu 4,5-krat višji zaradi uporabe **Swap** razdelka, kar pomeni, da potrebuje približno enako časa kot iskanje v ekvivalentnem kompaktnem priponskem drevesu. Iz tega testa se lahko sklepa, da za iskanje krajših vzorcev (vzorci do dolžine 10 znakov) je boljše uporabiti kompaktno priponsko drevo, sicer pa je boljše uporabiti priponsko drevo, če velikost delovnega spomina to omogoča ter če je število iskanih vzorcev dovolj veliko. S tem se lahko amortizira čas iskanja vzorcev, pri čemer je število vzorcev  $O(n)$ , kar zniža čas izgradnje drevesa za vsak vzorec na  $O(1)$ .

Iz testiranja izgradnje priponskega drevesa ter izvajanje poizvedb nad njim, za DNK sekvence je za iskanje v besedilih velikost do 8000 znakov boljše uporabiti priponsko drevo. Uporaba priponskega drevesa je tudi priporočljiva za daljše vzorce, kot so na primer vzorci dolžine 500 znakov. Pri tem mora biti število iskanih vzorcev dovolj veliko, da se amortizira čas, ki je potreben za izgradnjo priponskega drevesa (število vzorcev je  $O(n)$ ), sicer je za vsako priponsko drevo zgrajeno nad besedilom z vsaj 8000 znaki bolj priporočljivo, da se za iskanje uporablja kompaktno priponsko drevo. Uporaba kompaktnih priponskih dreves se izkaže boljša pri iskanju krajših vzorcev ter ko velikost priponskega drevesa presega velikost notranjega pomnilnika in zato mora bit shranjeno delno ali v celoti na **Swap** razdelku.

Druga primerjava je bila narejena s Cankarjevim romanom Na klancu [10]. Namen te primerjave je primerjati lastnosti priponskega drevesa nad večjo začetno abecedo, kot je na primer abeceda naravnega jezika. Ker slovenščina uporablja ne ASCII znake, je potrebno pred samim začetkom testiranja besedilo pred pripraviti. To je bilo storjeno tako, da so bili vsi ne ASCII znaki odstranjeni oziroma zamenjani z 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 zamenjani z osnovnim znakom, torej

na primer š postane s in é postane e. Pri tem so bile tudi odstranjene vse prazne vrstice ter ločila poglavij, ki so bila označena z '\*\*\*'. Odstranjeni so bili tudi vsi nevidni simboli, ki niso videni bralcu, prisotni v besedilu. Na ta način se je začetno besedilo znižalo iz dolžine 319843 znakov na 317803 znakov pred podaljševanjem. Naslednji korak je bila podaljšava besedila, ki podaljša velikost besedila iz 317803 na 25000000 znakov.

Uporabljeno besedilo uporablja abecedo velikosti  $52 + 1$  znak, pri čemer je originalna abeceda sestavljena iz slovenske abecede brez črk 'č', 'š', in 'ž' (bodisi v velikih črkah bodisi v malih črkah), ločil, presledkov in narekovajev. Pri tem vhodno besedilo zasede 25,851 MB delovnega spomina, kar je všteto v vse meritve velikosti različnih implementacij priponskih dreves.

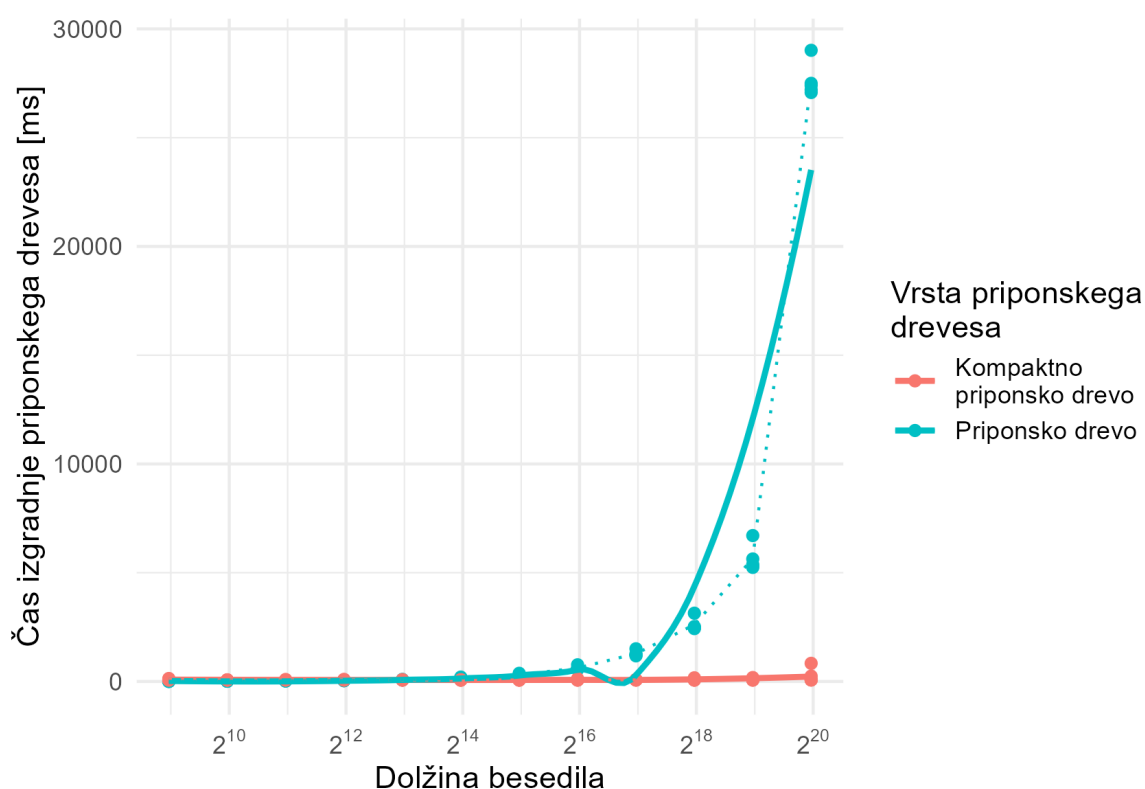


Slika 19: Graf velikosti priponskega drevesa izgrajenih iz besedil različnih velikosti. Vhodno besedilo je roman Na klancu.

Podobno kot za DNK sekvenco je bila izmerjena velikost, ki je potrebna za priponsko drevo na delavnem spominu. Rezultati meritve so prikazani na Sliki 19, kjer je z modro barvo prikazana velikost priponskega drevesa z rdečo barvo pa velikost kompaktne priponskega drevesa. Za to besedilo je velikost priponskega drevesa manjša ali približno enaka (manj kot 1,5-krat večja od kompaktne priponskega drevesa) kompaktnemu priponskemu drevesu, do besedila dolžine 4000 znakov. Velikosti priponskih dreves rastejo dokler ne dosežejo velikosti 3,9 GB za besedilo dolžine 1024000 znakov. Čeprav

ima priponsko drevo velikost delovnega pomnilnika, mora biti skoraj polovica drevesa shranjena na **Swap** razdelku, saj operacijski sistem zasede 1,76 GB delovnega spomina. Pri tem se zdi, kot da kompaktno priponsko drevo ohranja konstantno velikost v vsakem testiranju. To se zgolj zdi, saj kompaktno priponsko drevo potrebuje bistveno manj prostora kot največje testirano priponsko drevo. Velikost kompaktnega priponskega drevesa naraste iz 27,4 MB na 28,4 MB.

Iz Slike 19 je razvidno, da je priponsko drevo prostorsko bolj učinkovito za besedila, ki so krajša od 4000 znakov. Za vsa ostala daljša besedila je bolj prostorsko učinkovito uporabiti kompaktno priponsko drevo. To pa je zgolj eden od pogojev, kako izbrati primerno podatkovno strukturo.

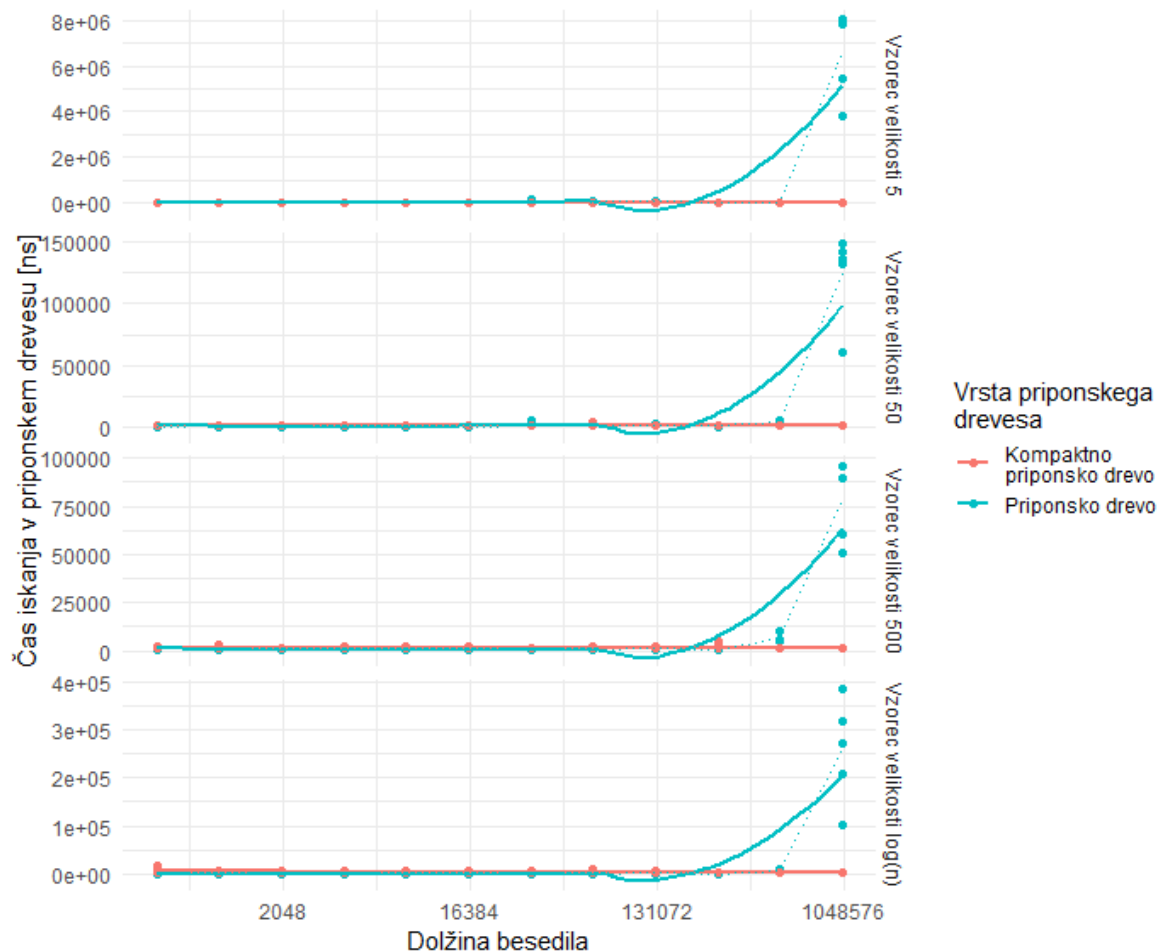


Slika 20: Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je roman Na klancu.

Naslednji pogoj, ki ga je potrebno upoštevati, je čas izgradnje priponskega drevesa, saj za je nekatere operacije bolj pomembno hitro izgraditi priponsko drevo, kot zasedati manj prostora na delovnem pomnilniku s priponskim drevesom. Na Sliki 20 je prikazan potreben čas za izgradnjo priponskega drevesa, ki je označen z modro barvo, ter čas potreben za izgradnjo kompaktnega priponskega drevesa, ki pa je označeno z rdečo barvo. Iz grafa je razvidno, da čas, ki je potreben za izgradnjo priponskega drevesa, je v vsakem koraku podvojen. Pri tem pa se razlikuje priponsko drevo nad besedilom velikosti 1024000 znakov, ki potrebuje 4,88-krat več časa za izgradnjo, kot priponsko

drevo nad besedilom predhodne testirane velikosti. Kompaktno priponsko drevo potrebuje bistveno manj časa za izgradnjo in iz slike se zdi, kot da bi potrebovalo konstantno časa. Kompaktno priponsko drevo potrebuje za izgradnjo kompaktne priponskega drevesa nad prvim testnim besedilom 89,8 milisekund in za izgradnjo nad največjim besedilom pa 259 milisekund.

Torej iz Slike 20 se lahko opazi, da je za besedila krajša od 8000 znakov čas izgradnje priponskega drevesa krajši kot za kompaktno priponsko drevo. Za daljša vhodna besedila pa se časovno izplača uporabiti kompaktno priponsko drevo.



Slika 21: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je roman Na klancu.

Čas izgradnje priponskega drevesa vpliva na hitrost iskanja vzorcev v besedilu tedaj, ko se ne išče dovolj vzorcev v besedilu, v tem primeru se čas izgradnje ne amortizira na konstanten dodaten čas iskanja posamičnega vzorca. V tem primeru je bolj pomemben čas iskanja vzorcev v besedilu. Zato se nazadnje lahko pogleda vpliv velikosti priponskega drevesa na potreben čas za izvršiti poizvedbo nad njimi. Na Sliki 21 so prikazani časi v nanosekundah potrebni za iskanje vzorcev, in sicer z modro barvo so predstavljeni časi za poizvedbe v priponskih drevesih, z rdečo barvo pa so predstavljeni časi



za poizvedbe v kompaktnem priponskem drevesu. Na sliki so predstavljeni rezultati za vse štiri velikosti vzorcev iskanih v priponskem drevesu: 5 znakov, 50 znakov, 500 znakov in  $\log n$  znakov, pri čemer je  $n$  dolžina besedila.

Na zgornjem grafu Slike 21 so prikazani rezultati iskanja vzorca dolžine 5 znakov. V priponskih drevesih je časovna zahtevnost iskanja vzorca konstantna skozi celotno izvajanje. Pri tem pa so bili izmerjeni 3 različni časi. Prvi izmerjen čas je približno 2,6 mikrosekund, ki je potreben za preveriti prisotnost vzorca v besedilih do 32000 znakov. V besedilih med 64000 in 512000 znakov je čas iskanja 15,5-krat večji, in sicer približno 45 mikrosekund. Najbolj verjetni razlog za ta skok v potrebnem času je dolžina niza, ki ga predstavljajo povezave med vozlišči. Nizi postanejo vedno krajši in zato je potrebno več časa za binarna iskanja v naslednje povezave. Drugi razlog za daljše iskanje je večja verjetnost prisotnosti vzorca v besedilu, saj imajo krajši vzorci v naravnem jeziku večjo verjetnost, da se ponovijo, ter vzorci, ki so narejeni iz podaljšanega besedila, so zagotovo prisotni v originalnem besedilu, razen če je vzorec lih zlepek dveh podnizov originalnega besedila. Zadnji izmerjen čas, pa je čas, ki je potreben za iskanje vzorca v besedilu dolžine 1024000 znakov. Le ta čas je 260,7-krat večji od predhodnega potrebnega časa, kar pomeni, da za najti vzorec je potrebno 6,59 milisekunde (6588 mikrosekunde). Razlogi za to povečavo v času so enaki kot za prejšnjo povečavo v času ter uporaba **Swap** razdelka, za shranjevanje priponskega drevesa.

Za razliko od priponskega drevesa, iskanje vzorca dolžine 5 v kompaktnem priponskem drevesu potrebuje konstantni čas, da se izvrši skozi celotno testiranje, kot je to lahko razvidno iz Slike 21. Iskanje v kompaktnem priponskem drevesu potrebuje približno enako časa kot iskanje v priponskih drevesih nad besedilo krajšim od 32000 znakov, za kar je potrebno približno 3 mikrosekunde. Iz tega sledi, da za besedila krajša 8000 znakov je iskanje v priponskem drevesu boljše, saj porabi manj prostora in manj časa, da se izgradi. Iskanje v besedilih do 32000 znakov je ekvivalentno natanko tedaj, ko je možno amortizirati čas izgradnje priponskega drevesa s količino iskanih vzorcev. V daljših primerih pa je boljše uporabiti kompaktno priponsko drevo za iskanje krajših vzorcev.

Naslednja testirana poizvedba je iskanje vzorcev dolžine 50 znakov, kar je možno videti na drugem grafu Slike 21. Opazi se, da je čas, ki je potreben za izvrši poizvedbo, konstanten skozi celotno izvajanje bodisi za kompaktno priponsko drevo, bodisi za priponsko drevo. Iskanje vzorcev dolžine 50 znakov s pomočjo priponskega drevesa potrebuje približno 1,5 mikrosekund. Z uporabo kompaktne priponskega drevesa pa je potrebnih približno 1,2 mikrosekunde. Pri tem je samo ena izjema, in sicer priponsko drevo, ki mora biti delno shranjeno na **Swap** razdelku. Uporaba **Swap** razdelka na trdem disku poveča čas iskanja vzorca za 27,3-krat, kar pomeni, da je potrebno 124 mikrosekund za preveriti, ali je vzorec prisoten v besedilu ali ni. Iz rezultatov

tega testiranja se lahko opazi, da ni časovne razlike med iskanjem vzorca dolžine 50 v besedilu z uporabo priponskega drevesa ali z uporabo kompaktnega priponskega drevesa. Torej, če je mogoče priponsko drevo shraniti v celoti na delovnem spominu in število iskanih vzorcev omogoča amortizacijo časa, ki je potrebnega za izgradnjo priponskega drevesa, je lahko le to uporabljeno za namene iskanja, sicer je bolje uporabiti kompaktno priponsko drevo.

Rezultati testiranja iskanja vzorca dolžine 500 znakov v besedilu, ki uporablja naravni jezik, so prikazani na tretjem grafu Slike 21. Podobno kot v predhodnih testiranjih je čas iskanja v obeh primerih konstanten. Priponsko drevo potrebuje približno 1 mikrosekundo za preveriti obstoj vzorca, kompaktno priponsko drevo pa potrebuje 1,5 mikrosekund. Pri tem pa obstaja ena izjema, in sicer priponsko drevo, ki je delno shranjeno na **Swap** razdelku. To priponsko drevo potrebuje 78,5 mikrosekund za izvesti poizvedbo, kar je 12,4-krat več čas kot v predhodnem priponskem drevesu ali pa 52,3-krat več čas kot povprečni čas ostalih poizvedb. Iz rezultatov tega testiranja je zaključek isti kot v predhodnem testiranju (iskanje vzorcev dolžine 50). To pomeni, da če se lahko priponsko drevo shrani v celoti na delovnem spominu in število iskanih vzorcev je dovolj veliko, da se lahko z vsako poizvedbo amortizira čas iskanja vzorcev, se lahko uporablja priponsko drevo, sicer je bolj priporočljivo uporabiti kompaktno priponsko drevo.

Zadnje izdelano testiranje je iskanje vzorcev dolžine  $O(\log n)$  v besedilu. Rezultati tega testiranja so predstavljeni na spodnjem grafu Slike 21. Podobno kot v primeru DNK sekvence so vzorci dolgi od 9 do 20 znakov. Skozi celotno izvajanje testiranja je čas poizvedbe v obeh primerih konstanten. Pri tem potrebuje poizvedba v priponskem drevesu približno 3,5 mikrosekund, v kompaktnem priponskem drevesu pa potrebuje ista poizvedba približno 4,5 mikrosekund. Izjema je priponsko drevo delno shranjeno v **Swap** razdelku, ki potrebuje 38,1-krat več časa od predhodnega priponskega drevesa ali 70,4-krat več časa od povprečnega časa poizvedbe. Torej potrebuje 257 mikrosekund za izvršiti poizvedbo. Podobno kot pri ostalih testiranjih nad besedilom v naravnem jeziku se lahko tudi iz teh rezultatov sklepa, da je priporočljivo uporabljati priponsko drevo za iskanje vzorcev v besedilu, če se lahko čas izgradnje le tega amortizira s količino iskanih vzorcev ter se lahko shrani celotno priponsko drevo v delovni spomin. Sicer pa je bolje, da se uporabi kompaktno priponsko drevo.

Iz vseh izvedenih testiranj nad priponskimi drevesi, ki so bila izgrajena nad besedilom v naravnem jeziku, se lahko sklepa, da je bolje uporabiti priponsko drevo za besedila do dolžine 4000 znakov, saj je potrebnega manj prostora in časa za izgradnjo drevesa, ter poizvedbe potrebujejo manj časa, da se izvršijo. Če pa je priponsko drevo zgrajeno nad daljšim besedilo in je lahko v celoti shranjeno v delovnem spominu, potem je potrebno preveriti ali je število iskanih vzorcev dovolj veliko, da vsaka poizvedba amortizira čas izgradnje besedila. Število vzorcev mora biti vsaj  $O(n)$ , da se lahko

amortizira čas izgradnje. Če ni dovolj vzorcev za to storiti, potem je bolje uporabiti kompaktno priponsko drevo. Le to omogoča malo počasnejše iskanje vzorcev, ampak za daljša besedila od 4000 znakov sta potreben čas za izgradnjo in prostor na pomnilniku bistveno nižja od prostora in časa izgradnje priponskega drevesa.

Iz rezultatov se lahko tudi opazi, da se priponskemu drevesu, ki je delno shranjeno na **swap** razdelku, bistveno poslabša čas potrebnem za izgradnjo in iskanje vzorcev v besedilu. Po tem takem ni priporočljivo uporabljati priponska drevesa, ki morajo biti shranjena izven delovnega spomina.

Pri primerjavi rezultatov obeh testiranj, testiranje nad DNK sekvenco in testiranje nad besedilom iz naravnega jezika, se lahko opazi, da so si rezultati zelo podobni. Edina bistvena razlika med obema testiranjema je skok v potrebnem času pri iskanju daljših vzorcev ter vzorcev dolžine  $O(\log n)$  v besedilu, ki je zapisano v naravnem jeziku. Najbolj verjeten razlog je prisotnost vzorca v besedilu, ki poveča časovno zahtevnost iskanja. Torej iz testiranja se lahko ugotovi, da ni nobene razlike med časom potrebnim za poizvedbo in izgradnjo ter prostorsko zahtevnostjo priponskega drevesa (oziroma kompaktnega priponskega drevesa), glede na vhodno besedilo priponskega drevesa.

Obstaja pa izmerljiva razlika med priponskimi drevesi, ki so v celoti shranjeni v delovnem spominu, ter tistimi, ki so deloma shranjeni na **Swap** razdelku. Izmerjena razlika se pojavi tudi v primerjavi, ki so jo izdelali Välimäki idr. [3]. Izmerjena razlika je skladna z razliko v času branja zaporednih podatkov med trdim diskom in notranjim spominom, ki je približno 7-krat počasnejši, kar je izmeril Jacobs [21]. Iz ugotovitev od Jacobsa [21] in Välimäki idr. [3] je lahko pojasnjena velika časovna zahtevnost pri izgradnji priponskega drevesa velikost 2048000 znakov.

## 7 ZAKLJUČEK

Namen magistrske naloge je bila predstavitev podatkovne strukture kompaktno priponsko drevo ter primerjava le te s podatkovno strukturo priponsko drevo. Obe podatkovni strukturi sta bili primerjani med seboj, bodisi teoretično bodisi empirično.

Pred samo primerjavo obeh podatkovnikovih struktur, je bila vsaka podatkovna struktura predstavljena. Predstavitev ni vsebovala samo predstavitev implementacije podatkovne strukture, ampak tudi predstavitev algoritmov za izgradnjo podatkovne strukture. Sama teoretična primerjava se je osredotočila na tri različne primerjave, in sicer na primerjavo osnovnih operacij, primerjavo osnovnih poizvedb ter primerjavo prostorske zahtevnosti in časovne zahtevnosti algoritmov za izgradnjo. Iz primerjave osnovnih operacij in primerjave osnovnih poizvedb je razvidno, da imajo nekatere operacije ter poizvedbe različen čas izvajanja v kompaktnem priponskem drevesu, in sicer se potreben čas pri osnovnih operacijah lahko dvigne za  $O(t_{SA})$ -krat ali  $O(t_\Psi)$ -krat. Pri poizvedbah se čas dvigne zgolj za  $O(t_\Psi)$  pri poizvedbi *prisotnost(vzorec)*, pri ostalih poizvedbah pa se potrebni čas zmanjša za  $O(n)$ . Časa  $O(t_\Psi)$  in  $O(t_{SA})$  sta odvisna od implementacije kompaktne priponskega polja, ki je uporabljeno v kompaktnem priponskem drevesu. Če je kompaktno priponsko drevo implementirano s pomočjo časovno najbolj učinkovite implementacije kompaktne priponskega polja, potem je čas  $t_\Psi = O(1)$  in  $t_{SA} = O(\log^\epsilon n)$ , kar pomeni da vse osnovne operacije, ki so bile za  $O(t_\Psi)$ -krat počasnejše, ohranijo isto časovno zahtevnost, kot jo potrebujejo v priponskem drevesu.

Glavna razlika med priponskim drevesom in kompaktnim priponskim drevesom, je v časovni zahtevnosti izgradnje ter prostorski zahtevnosti. Priponsko drevo potrebuje  $O(n)$  povezav, kar pomeni, da potrebuje  $O(n \log n)$  bitov ali  $O(nw)$  bitov, če se uporabljajo sistemski naslovi. Kompaktno priponsko drevo pa potrebuje  $|CSA| + 6n + o(n)$  bitov, kar v obeh predstavljenih implementacijah kompaktne priponskega polja ohrani prostorsko zahtevnost  $O(n)$  bitov. Časovna zahtevnost izgradnje priponskega drevesa se dvigne iz  $O(n)$  za priponska drevesa na  $O(n \log n)$  za kompaktna priponska drevesa, kar pa 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 zahtevnosti ter v različni časovni zahtevnosti operacij, ki so bile predhodno predstavljene. Empirična primerjava je bila opravljena nad zaporedjem genov ter nad besedilom v naravnem jeziku (Slovenščina). Empirična primerjava pa je merila časovno zahtevnost poizvedbe

*prisotnost(vzorec)* nad besedili različnih velikosti ter vzorcev različnih dolžin, časovno zahtevnost izgradnje priponskega drevesa različnih velikosti ter prostorsko zahtevnost le teh priponskih dreves. Z empirično primerjavo se lahko potrdijo teoretične časovne razlike operacij in poizvedb med priponskimi drevesi in kompaktnimi priponskimi drevesi. Pri tem pa se tudi opazi razlika med časovno zahtevnostjo operacij, ko je priponsko drevo v celoti shranjeno na delovnem spominu ter ko je del drevesa shranjen na **Swap** razdelku. Te razlike ni mogoče opaziti na kompaktnih priponskih drevesih, saj dolžina besedila je prekratka, da bi bilo potrebno shraniti kompaktno priponsko drevo na **Swap** razdelku. Pri tem se tudi lahko opazi, da je smiselno uporabljati kompaktna priponska drevesa za besedila, ki imajo dolžino vsaj 8000 znakov, saj takrat kompaktna priponska drevesa potrebujejo manj prostora ter manj časa, da se izgradijo.

Rezultati testiranja so bili izdelani na računalniku z zgolj 4 GB delovnega pomnilnika in 8 GB **Swap** razdelka, kar je relativno malo spomina, saj imajo novi osebni računalniki vsaj 8 GB delovnega spomina ter dodaten **Swap** razdelek in procesorji za strežnike podpirajo več 100 GB delovnega spomina. Torej se pojavi vprašanje ali so kompaktne podatkovne strukture sploh še potrebne, saj imajo trenutni računalniki na razpolago dovolj delovnega spomina za shraniti celotno priponsko drevo. Po mojem mnenju so še vedno potrebne, saj omogočajo shranjevanje in iskanje po večjem številu priponskih dreves hkrati. Iskanje vzorcev v večjem številu priponskih dreves na enkrat je mogoče, saj večina procesorjev podpira izvajanje večjega števila procesov na enkrat. Računalnik, na katerem je bilo izdelano testiranje, omogoča izvajanje do 4 procesov na enkrat, saj ima 2 jedri in 4 niti.

Nadaljnje raziskave bi se morale osredotočiti na implementacijo kompaktne priponskega polja, ki zniža obe časovni zahtevnosti  $t_{SA}$  in  $t_{\Psi}$  na  $O(1)$ . Na ta način bi se znižala razlika med kompaktnimi priponskimi drevesi ter priponskimi drevesi, kar bi omogočalo vse prednosti priponskega drevesa ter vse prednosti kompaktnih podatkovnih struktur.

## 8 LITERATURA IN VIRI

- [1] E. UKKONEN, On-line construction of suffix trees. *Algorithmica* 14 (1995) 249–260. (*Citirano na straneh 21, 25 in 28.*)
- [2] K. SADAKANE, Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems* 41 (2007) 589–607. (*Citirano na straneh 43, 44, 45, 47, 48 in 49.*)
- [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 45, 55, 64, 70, 71 in 81.*)
- [4] E. M. MCCREIGHT, A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery* 23 (1976) 262–272. (*Citirano na straneh 21, 22 in 28.*)
- [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, 55 in 64.*)
- [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, 12, 15, 16, 45, 46, 47, 48, 50, 51, 52, 53, 55, 62 in 64.*)
- [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 17.*)
- [8] D. GUSFIELD, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997. (*Citirano na strani 17.*)
- [9] *Pizza&Chili Corpus – Compressed Indexes and their Testbeds*, <https://pizzachili.dcc.uchile.cl/>. (Datum ogleda: 8. 10. 2024.) (*Citirano na straneh 66, 69 in 71.*)
- [10] I. CANKAR, *Na Klancu*, Genija, 2012. (*Citirano na straneh 66, 69 in 75.*)

- [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 straneh 68 in 70.*)
- [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 68.*)
- [13] E. MILLER, *Genome — Knowledge Hub*, <https://www.genomicseducation.hee.nhs.uk/genotes/knowledge-hub/genome>. (Datum ogleda: 30. 10. 2024.) (*Citirano na strani 43.*)
- [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 34, 35, 39 in 51.*)
- [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 44.*)
- [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 45, 49, 57 in 64.*)
- [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 45, 49 in 57.*)
- [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 49.*)
- [19] D. HAREL in R. E. TARJAN, Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing* 13 (1984) 338–355. (*Citirano na strani 61.*)
- [20] KOUTE, *koute/bytehound: A memory profiler for Linux.*, <https://github.com/koute/bytehound>. (Datum ogleda: 30. 10. 2024.) (*Citirano na strani 65.*)
- [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 81.*)

- [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 33 in 40.)
- [25] P. KO in S. ALURU, Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms* 3 (2005) 143-156. (Citirano na strani 40.)
- [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 34, 35, 36 in 37.)
- [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 35.)