UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Magistrsko delo

Kompaktna priponska drevesa

(Compact Suffix Tree)

Ime in priimek: *Jani Suban*
Študijski program: *Računalništvo in informatika, 2. stopnja*
Mentor: *prof. dr. Andrej Brodnik*
Somentor: *izr. prof. dr. Rok Požar*

Koper, 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

1 UVOD **1**
1.1 STRUKTURA 1

2 OZNAKE, DEFINICIJE IN OSNOVNE OPERACIJE **3**
2.1 OZNAKE IN OSNOVNE DEFINICIJE 3
2.2 ENIŠKI ZAPIS 4
2.3 MODEL RAČUNANJA 4
2.4 BITNO POLJE 5
2.5 KOMPAKTNA PREDSTAVITEV DREVES 11
2.5.1 Zaporedje eniških zapisov stopenj vozlišč po plasteh 12
2.5.2 Zaporedje eniških zapisov stopenj vozlišč v globino 14
2.5.3 Uravnoteženi oklepaji 16

3 PRIPONSKA DREVESA **19**
3.1 POIZVEDBE 20
3.2 GRADNJA 23
3.2.1 Naivna metoda 23
3.2.2 Izboljšana naivna metoda 26
3.2.3 Ukkonenov algoritem 28

4 PRIPONSKO POLJE **33**
4.1 POIZVEDBE 34
4.2 GRADNJA 38
4.3 SIMULACIJA OPERACIJ PRIPONSKEGA DREVESA 39

5 KOMPAKTNA PREDSTAVITEV PRIPONSKEGA DREVESA **45**
5.1 POIZVEDBE 53
5.2 GRADNJA 55

6 EMPIRIČNA EVALVACIJA OPERACIJI NAD PRIPOSKIMI DREVESI **59**
6.1 EKASIPERIMENTALNO OKOLJE 59
6.1.1 Stroj, operaciski sistem in metoda testiranja 59

6.1.2 Testni primeri 63
6.2 PRIMERJAVA 65
6.2.1 Gradnja 65
6.2.2 Poizvedbe 66
6.3 RAZPRAVA 66

7 ZAKLJUČEK **70**

8 LITERATURA IN VIRI **72**

Kazalo preglednic

1 Implementacija operacij drevesa v LOUDS. 13
2 Implementacija operacij drevesa z DFUDS 15
3 Implementacija operacij drevesa z BP 17

4 Primerjava besedil, ki bodo uporabljena za primerjavo različnih implementacij priponskih dreves 64

Kazalo slik in grafikonov

1 Primer dodatne podatkovne strukture za operacijo $rank_1$ v konstantnem času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine $w = 4$. Pri tem se je izbralo, da je število $k = 4$. 7
2 Primer dodatne podatkovne strukture za operacijo $izbira_1$ v $O(\log \log n)$ času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine $w = 4$. Zaradi majhnosti primera velika vedra, ki vsebujejo vsaj 10 bitov namesto $s \log^2 b = 4 \log^2 36$ bito, kar se zaokroži na 100 bito. Velika vedra so označena v bitem polju s sivo barvo. 8
3 Primer predstavitve drevesa z metodo LOUDS za priponsko drevo besede »KOKOŠ$«. 13
4 Primer predstavitve drevesa z metodo DFUDS za priponsko drevo besede »KOKOŠ$«. 15
5 Primer predstavitve drevesa z metodo BP za priponsko drevo besede »KOKOŠ$«. 17

6 Primer priponskega drevesa nad besedo »KOKOŠ$«. 20
7 Načini dodajanja novih znakov v priponsko drevo. Na sliki kvadratki predstavljajo liste drevesa, krogci pa predstavljajo notranja vozlišča drevesa. 24
8 Primer izgradnje priponskega drevesa z uporabo Naivne metode za besedo »KOKOŠ$«. 25
9 Primer izgradnje priponskega drevesa z uporabo Izboljšane naivne metode za besedo »KOKOŠ$«. 27
10 Primer izgradnje priponskega drevesa z uporabo Ukkonenovaga algoritma za besedo »KOKOŠ$«. 29

11 Primer priponskega drevesa in priponskega polja z dodatnim LCP poljem nad besedo »KOKOŠ$«. 33
12 Prikaz razmerja med P in priponsama 35
13 Primer *L-LCP* in *R-LCP* polji za priponskega polja nad besedo »KOKOŠ$«. 37
14 Primer LCP intervalnega drevesa nad priponskim poljem in LCP poljem besede »KOKOŠ$« ter njegova predstavitev s tabelo. Na sliki je dodano tudi priponsko drevo besede »KOKOŠ$«. 42

15 Primer komponent kompaktnega priponskega drevesa nad besedo »KOKOŠ$«. 52

16 Posnetek zaslona upravljalnika opravil Htop med izgradnjo priponskega drevesa za besedilo dolžine 2048000 znakov. 63
17 Graf velikosti indeksov izgrajenih iz besed različni velikosti. Vhodna beseda je DNK sekvenca. 65
18 Graf velikosti indeksov izgrajenih iz besed različni velikosti. Vhodna beseda je roman Na klancu. 66
19 Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je DNK sekvenca. 67
20 Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je roman Na klancu. 68
21 Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je DNK sekvenca. . . 69
22 Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je roman Na klancu. 69

Seznam kratic

angl.	Angleščina
slo.	Slovenščina
idr.	in drugi
DNK	Deoksiribonukleinska kislina
KMP	Knuth–Morris–Pratt
ST	angl. *Suffix Tree* (slo. Priponsko drevo)
rmM	angl. *range minimum-maxsimum*
rmq	angl. *range minimum querry*
rMq	angl. *range maxsimum querry*
LOUDS	angl. *Level Order Unary Degree Sequence* (slo. Zaporedje eniških zapisov stopenj vozlišč po plasteh)
DFUDS	angl. *Depth Frst Unary Degree Sequence* (slo. Zaporedje eniških zapisov stopenj vozlišč v globino)
BP	angl. *Balanced Parentheses* (slo. Uravnoteženi oklepaji)
SA	angl. *Suffix Array* (slo. Priponsko Polje)
LCP	angl. *Longest Common Preffix* (slo. Najdaljša skupna predpona)
BWT	angl. *Burrows–Wheeler Transform* (slo. Burrows–Wheelerjeva preslikava)
CST	angl. *Compressed Suffix Tree* (slo. Kompaktno priponsko drevo)
CSA	angl. *Compressed Suffix Array* (slo. Kompaktno priponsko polje)
CSV	angl. *Comma-separated values*
GCC	angl. *GNU C Compiler* (slo. GNU C Prevajalnik)
SDSL	angl. *Succinct Data Structure Library* (slo. Knjižnica kompaktnih podatkovnih struktur)
ASCII	angl. *American Standard Code for Information Interchange* (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 besed je iskanje vzorcev v njih. V procesiranju naravnih jezikov vlogo vzorca pogosto prevzame iskana beseda naravnega jezika in vlogo vhodne besede pa prevzame besedilo, v katerem želimo iskati besedo. Medtem ko v bioinformatiki vlogo vzorca prevzame specifični gen ali druga DNK sekvenca, pri tem pa je vhodna beseda 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 sprehoda iz korena proti listom priponskega drevesa, zgrajenega nad besedo. Peter Weiner je kot prvi predstavil priponska drevesa leta 1973 [5] in jih uporabil v algoritmu za iskanje vzorcev v besedi. Časovna zahtevnost podanega algoritma za iskanje vzorcev v besedi je $O(m)$. Velikost priponskega drevesa je sorazmerna z dolžino vhodne besede, torej pri velikih besedah preraste velikost delovnega pomnilnika, kar močno vpliva na hitrost iskanja. Rešitev te težave je kompaktna predstavitev priponskega drevesa [6].

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

1.1 STRUKTURA

Magistrska naloga je razdeljena na pet delov. V drugem poglavju bodo predstavljene notacije 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 sprotno (angl. *on-line*) zgradi priponsko drevo v času $O(n)$ ter implementacija poizvedb nad vhodno besedo s pomočjo priponskega drevesa. Zatem sledi v četrtem poglavju predstavitev alternativnega indeksa besedila, priponsko polje (angl. *Suffix Array* oziroma SA). Predstavljena bo tudi metoda izgradnje priponskega polja v $O(n)$ času ter implementacija poizvedb s priponskim poljem.

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

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

2 OZNAKE, DEFINICIJE IN OSNOVNE OPERACIJE

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

2.1 OZNAKE IN OSNOVNE DEFINICIJE

Skozi magistrsko nalogo bo Σ predstavljala abecedo. Abeceda je množica znakov $\Sigma = \{\sigma_1, \ldots, \sigma_a\}$. Vhodna beseda T dolžine n znakov, iz katere bomo zgradili priponsko drevo, priponsko polje ali kompaktno priponsko drevo, je sestavljena iz znakov abecede Σ, torej je $T = \Sigma^n$. Na podoben način P predstavlja iskani vzorec dolžine m znakov, katerega se želi preveriti prisotnost v T. Tako kot beseda T je tudi vzorec P sestavljen iz znakov abecede Σ, torej je $P = \Sigma^m$.

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

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

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

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

Za razliko od nizov in podnizov so znaki iz abecede Σ 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, ko se govori na splošno o listih, sicer pa bodo označeni z l_i, pri čemer i označuje zaporedno število lista od leve proti desni.

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

2.2 ENIŠKI ZAPIS

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

Eniški zapis bo uporabljen pri kompaktni predstavitvi dreves. Razlog za uporabo eniškega zapisa števil namesto dvojiškega zapisa števil 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 MODEL RAČUNANJA

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 potreno narediti analizo za vsako arhitekuro posebaj. 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 pomnilnika 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 neskončno velik pomnilnik za razliko od besednega RAM, ki pa omeji velikost pomnilnika na U naslovov in posledično je velikost ene pomnilniške besede na $w = \log U$ bitov, kar je velikost enega naslova [6, 22, 23]. Pri tem se lahko definira tudi »cela števila« (angl. *Integer*). »Cela števila« so definirana kot podmnožica celih števil, ki so lahko predstavljena v dvojiškem zapisu z eno računalniško besedo [6].

2.4 BITNO POLJE

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

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

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

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

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

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 jo definiramo nad bitnim poljem B velikosti b, je $rang_v(B, i)$. Operacija vrne število bitov z vrednostjo $v \in \{1, 0\}$ v B do vključno položaja i. V nadaljevanju bo operacija $rang(B, i)$ predstavljala $rang_1(B, i)$. To je možno, saj velja sledeča zveza med $rang_0(B, i)$ in $rang_1(B, i)$:

$$rang_1(B, i) = i - rang_0(B, i). \tag{2.1}$$

Operacijo se lahko naivno implenetira s štetjem bitov z vrednostjo 1 v bitenem polju, za kar je potrebno $O(b)$ časa. Štetje bitov z vrednostjo 1 se v angl. imenuje *population count* (v nadaljevanju označeno `popcount`), ki se izračuna v konstantnem času za eno pomnilniško besedo [27, 29]. Zaradi uporabnosti te funkcije je le ta že implementirana v različnih modernih arhitekturah procesorjev in je zato možno uporabiti strojno operacijo. Operacijo rang se lahko pospeši na konstantni čas, pri tem pa se potrebuje dodatne podatkovne strukture, ki zasedejo $o(b)$ bitov. Pri tem ni potrebno zgraditi dodatnih podatkovnih strukur za obe različici ranga, saj relacija iz enakosti (2.1) omogoča izgradnjo podatkovne strukture zgolj za operacijo $rang_1$.

Pomožna struktura shranjuje range različnih elementov bitnega polja B v polju R. Naivni pristop bi shranil v polju R range vseh elementov B-ja. Problem tega pristopa je prevelika prostorska zahtevnost, saj je zaželjeno, da je R čim manjši in pomožnosti ni večji od $o(b)$ bitov. Predlagana rešitev potrebuje $b \log r_1 = O(bw)$ dodatnih bitov, pričemer 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 $R[i] = rang_1(B, is)$, pri čemer $0 \leq i \leq \left\lfloor \frac{b}{s} \right\rfloor$. Na ta način je operacija rang implementiran kot

$$rang_1(B, i) = R\left[\left\lfloor \frac{i}{s} \right\rfloor\right] + \texttt{popcount}\left(B\left[\left\lfloor \frac{i}{s} \right\rfloor s, i\right]\right).$$

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

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

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

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

Slika 1: Primer dodatne podatkovne strukture za operacijo $rank_1$ v konstantnem času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine $w = 4$. Pri tem se je izbralo, da je število $k = 4$.

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

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 $j = rang_v(B, izbira_v(B, i))$. Pri tem pa ne obstaja 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 $izbira_1(B, i)$, saj se lahko $izbira_0(B, i)$ implementira na podoben način [6].

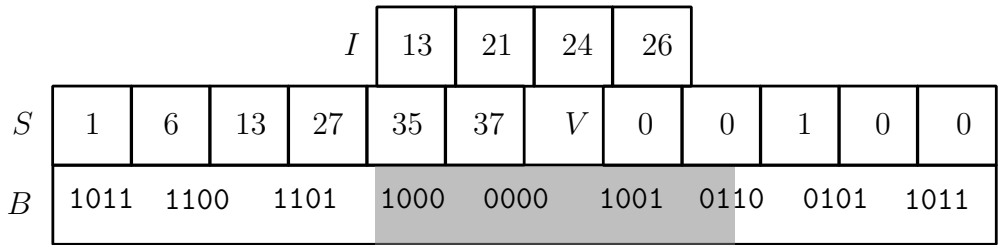
Ko operacija izbira ni ključna pri reševanju problemov, se lahko uporabi binarno iskanje v poljih R in R', ki sta del podatkovne strukture za rank, za kar se potrebuje $O(\log b)$ časa. Z binarnim iskanjem se najde območje dolžine k, pri čemer je potrebno še dodatnih k preiskav, da se najde natančno 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 $S[i] = izbira_1(B, i \cdot s + 1)$, pri čemer $S[\lceil \frac{r_1}{s} \rceil] = b + 1$. Polje S potrebuje $w(\lceil \frac{r_1}{s} \rceil + 1)$ bitov. Ko je $s = w^2$, potem podatkovna struktura potrebuje $w(\lceil \frac{r_1}{w^2} \rceil + 1) = o(r_1) = o(b)$ bitov [6].

Vedra niso enako velika, zato se lahko razdelijo na velika vedra in mala vedra, pri čemer je vedro veliko natanko tedaj, ko je večje kot $s \log^2 b$ bitov, sicer je malo vedro. Pri tem je potrebno shraniti velikost vedra v bitno polje V, kjer $V[i] = 1$, če i-to vedro je veliko, sicer je $V[i] = 0$. Za velika vedra se 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 kot

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

kjer je $x = ((j-1) \mod s) + 1$ in k je relativni položaj x-tega bita z vrednostjo 1 na intervalu $B\left[S\left[\lceil \frac{j}{s} - 1 \rceil\right], S\left[\lceil \frac{j}{s} \rceil - 1\right]\right]$. Vrednost k je izračunana s pomočjo bisekcije na podatovni strukturi za rank, ter šteje bitov na zadnjem intervalu [6]. Podatkova struktura za izbiro je prikazana na Sliki 2.



Slika 2: Primer dodatne podatkovne strukture za operacijo $izbira_1$ v $O(\log\log n)$ času. Pri tem je bitno polje razdeljeno na posamične pomnilniške besede dolžine $w = 4$. Zaradi majhnosti primera velika vedra, ki vsebujejo vsaj 10 bitov namesto $s\log^2 b = 4\log^2 36$ bito, kar se zaokroži na 100 bito. Velika vedra so označena v bitem polju s sivo barvo.

Polje I mora shraniti $s\lceil \log b \rceil$ 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 $s' \log\left(s(\log b)^2\right) = O((\log\log b)^3) = o(b)$ 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 polj B, pri čemer vsak sprehod traja $O(b)$ časa. Pri tem je ves potrebni spomin že predhodno dodeljen.

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

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

Operacija predhodnik elementa y najde največji indeks $i < y$, za katerega velja, da je $B[i] = v$. 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 operacijo naslednik. Operacija naslednik elementa y najde najmanjši indeks $i > y$, za katerega velja, da je $B[i] = v$. Pri tem je operacija implementirana kot

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

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 polj B).

Višek. 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 je izračunan kot:

$$višek(B, i) = rang_0(B, i) - rang_1(B, i) = 2\, rang_0(B, i) - i.$$

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

$$višek(B, i, j) = višek(B, j) - višek(B, i - 1).$$

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

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

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

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

Vse štiri operacije temeljijo na sprehodu po rmM-drevesu, torej bo sprehod predstavljen zgolj za operacijo $rmq(B, i, j)$ na intervalu bitnega polja $B[i : j]$. Operacija $rmq(B, i, j)$ je izvedena v dveh korakih. Prvi korak je iskanje vrednosti najmanjšega viška na intervalu $B[i : j]$. Ker se lahko i nahaja sredi vedra bitov, se 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 starša, torej $v \leftarrow (v-1)/2$. Sicer pa je potrebno preveriti, ali je desni brat tudi v intervalu, kar se lahko preveri, tako da preverimo, če trditev $\lfloor l/2^{\lfloor \log l \rfloor - \lfloor \log u \rfloor} \rfloor \neq u$ drži, potem je vozlišče u tudi znotraj intervala $B[i : j]$. Če je desni brat v intervalu preverimo, ali smo našli nov najmanjši višek na intervalu $(m > d + rmM[v+1].m)$. Če smo ga našli, potem $m \leftarrow d + rmM[v+1].m$. Nato pa se še popravi višek intervala na $d \leftarrow d + rmM[v+1].e$. Zatem pa se premaknemo v starša, torej $v \leftarrow v/2$ [6].

Ko desni brat ni v intervalu, torej ne velja prejšnji pogoj, pa nadaljujemo s spreho-

dom 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 $rmM[v].m$, ampak se išče prvo pojavitev viška, ki je enaka m. Ta isti postopek se lahko uporabi tudi za preostale tri operacije, pri čemer pa operacija $rMq(B, i, j)$ uporablja namesto polja $rmM[v].m$ polje $rmM[v].M$ in relacija manjše postane večje in obratno. Operaciji $minIzbira(B, i, j, t)$ in $minŠtetje(B, i, j)$ pa uporabljata polje $rmM[v].n$ [6].

Tako implementirane operacije potrebujejo $O(\log b)$ časa, da se izvršijo. Začetno in končno štetje bitov potrebuje $O(a) = O(1)$ čas. Ker je rmM-drevo dvojiško in polno (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].

2.5 KOMPAKTNA 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 referenc na njih. V vsakem vozlišču je shranjena vrednost, ki jo predstavlja vozlišče, ter referenca na svoje otroke. Drevo z n-timi vozlišči potrebuje $O(n \log n)$ bitov za shraniti topologijo drevesa (v praksi potrebuje $O(nw)$ bitov). Kompaktna predstavitev topologije dreves zniža prostorsko zahtevnost

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

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

Definicija 2.3. Podatkovne struktura drevo mora podpirati naslednje operacije:

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

Iz definicije se lahko opazid, da večina operacij je lahko implementiranih zgolj s operacijo $Otrok(v, i)$ ter operacio $starš(v)$, s katerimi le lahko sprehajamo po drevesu. S pomočjo operacij iz definicije je možno implementirati ostale operacije na drevesih, ki so bolj specifične za posamično implementacijo drevesa, na primer `vstavi`, `izbriši`, `najmanjši_element` (v kopici) in ostale.

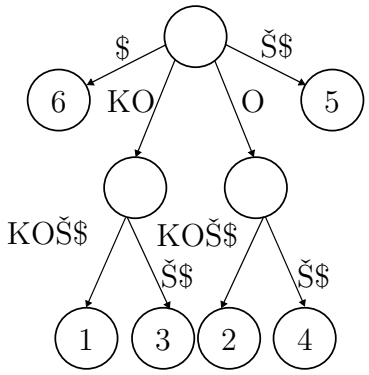
2.5.1 Zaporedje eniških zapisov stopenj vozlišč po plasteh

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

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

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

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



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

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

Tabela 1: Implementacija operacij drevesa v LOUDS.

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

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

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

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

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

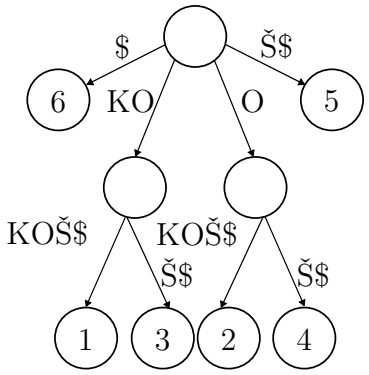
Vhod: Bitno polje B, vozlišča v in w
Izhod: Vozlišče u

```
1 while v ≠ w do
2     če v > w potem
3         v ← starš(v)
4     sicer
5         w ← starš(w)
6 vrni v
```

2.5.2 Zaporedje eniških zapisov stopenj vozlišč v globino

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

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



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

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

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

Tabela 2: Implementacija operacij drevesa z DFUDS

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

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

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

Vse operacije, ki temeljijo na operaciji *rang* in *izbira*, se izvršijo v $O(1)$ času. Operacije, ki temeljijo na rmM-drevesu, pa potrebujejo $O(\log n)$ časa. Pri tem so potrebne dodatne podatkovne strukture za $izbira_1$, $izbira_0$, *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.

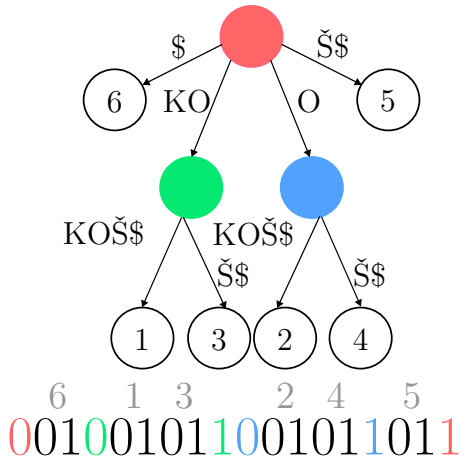
Za lažjo implementacijo dodatnih operacij nad listi je mogoče ustvariti dve dodatni podatkovni struktur za $rang_{00}$ in $izbira_{00}$ v konstantnem času, ki omogočata iskanje in indeksiranje listov v drevesu. Podatkovni strukturi sta implementirani na podoben način kot podatkovni strukturi za $rang_0$ 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.

2.5.3 Uravnoteženi oklepaji

Zadnji način zapisa topologije drevesa je zapis z zaporedjem uravnoteženih 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 uravnoteženih oklepajev, je prikazan na Sliki 5. Na sliki so vsa notranja vozlišča obarvana z različnimi barvami za lažje prepoznavanje le teh v zaporedju uravnoteženih oklepajev. Ker so listi oštevilčeni, so ta števila tudi zapisana nad vsakim listom v zaporedju uravnoteženih oklepajev [6].

Torej je vsako vozlišče predstavljeno kot par '(' in ')'. Tako je mogoče definirati sledeče operacije nad oklepaji: *odpri*, *zapri* ter *oklepa*. Operacija $odpri(B, i)$ vrne položaj '(', ki odpre ')' na i-tem mestu v B. Operacija $zapri(B, i)$ pa v tej notaciji predstavlja ')', ki zapra '(' na i-tem mestu B. Operacija $oklepa(B, i)$ pa vrne položaj j od '(' v bitem polju B, pri čemer velja, da $j < i < zapri(B, j)$. Z uporabo operacije *višek* operacija *OklepaBi* 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 $o(n)$ do-



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

datnih bitov in operaciji se izvršita v konstantnem času.

Tabela 3: Implementacija operacij drevesa z BP

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

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

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

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

3 PRIPONSKA DREVESA

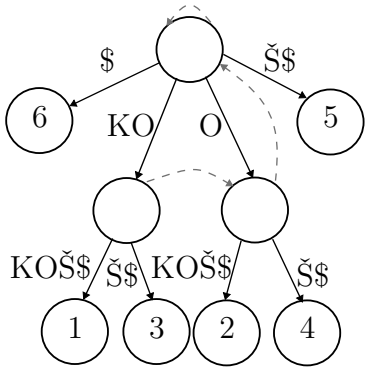
Priponska drevesa so posebna implementacija številskega drevesa, pri čemer vsak list predstavlja posamezno pripono besede. Na ta način priponsko drevo ne predstavlja zgolj vhodne besede T dolžine n, ki je sestavljeno iz znakov abecede Σ in je shranjena v pomnilniku kot polje črk `T[1:n]`, ampak zakodira tudi strukturo besede. 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. KMP algoritmom 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].

Nad priponskim drevesom lahko definiramo funkcijo $beseda(v)$. Funkcija vrne niz, ki je pridobljen s stikom podnizov na povezavah na poti od korena do vozlišča v. Torej priponsko drevo nad besedo T, ki je niz nad abecedo Σ, 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, ki je shranjena v polju $indeks$,
2. vsako notranje vozlišče, razen korena, ima vsaj dva otroka,
3. vsaka povezava predstavlja neprazni podniz besede T,
4. ne obstajata povezavi, ki se začneta v istem vozlišču in z istim znakom,
5. za vsak list l velja, da je $beseda(l) = T[l.ideks :]$.

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

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

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

3.1 POIZVEDBE

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

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

Osnovna poizvedba nad besedo T je *prisotnost*(T, P), saj sta ostali dve poizvedbi nadgradnji le te. Osnovna ideja iskanja vzorcev s priponskimi drevesi je obstoj vzorca P na začetku vsaj ene pripone besede T natako tedaj, ko je P prisoten v T. Oziroma vzorec $P[1 : m]$ je prisoten v besedi T natako tedaj, ko obstaja pripona $T[i : n]$, za katero velja $P = T[i : i + m]$ (P je predpona pripone $T[i : n]$).

Listi priponskega drevesa predstavljajo pripone besede T, zato je poizvedba *prisotnost*(T, P) z uporabo priponskega drevesa implementirana s sprehodom iz *korena* drevesa proti listom. Ker vsaka povezava predstavlja podniz besede $T[k : p]$, ki je lahko shranjen kot par indeksov k in p, je potrebno preveriti, ali se P ujema s $T[k : p]$. Če se vzorec ne ujema s podnizom, potem vzorec ni prisoten v besedi in zato poizvedba vrne *false*. Ko pa se P ujema s podnizom, potem je vzorec prisoten v besedi in zato poizvedba vrne *true*. Tako predstavljen način iskanja deluje zgolj za vzorce, ki niso daljši od $p-k$ znakov. Če pa je vzorec P daljši, se najprej pogleda prvih $p-k$ znakov. Če se podniza ujemata, se nadaljuje z iskanjem na naslednji povezavi, ki se začne v vozlišču v, do katerega smo prišli po povezavi $T[k : p]$ na poti proti listom. Pri tem si je potrebno zapomniti, koliko znakov smo že pregledali, in to označimo z o. V vsakem koraku povečamo o za $p-k$ ter perverimo, 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 Σ ne spreminja skozi postopek izgradnje in poizvedb.

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

Izrek 3.3. *Poizvedba prisotnost*(T, P) *implementiran s priponskim drevesemo potrebuje* $O(m)$ *časa in* $O(n)$ *prostora.*

Za primer iskanja vzememo priponsko drevo na Sliki 6, ki predstavlja besedo T =»KOKOŠ«. V besedi želimo preveriti prisotnost vzorca P_1 = »KOŠ«, ki je prisoten v besedi, ter vzorca P_2 = »KOT«, ki pa ni prisoten v besedi. Pri tem predpostavimo, da ima vsako vozlišče fiksno polje kazalcev na otroke (vsaka črka abecede ima eno celico v polju). Iskanje se začne v vozlišču `koren`. Preveri se, ali velja `koren.otroci['K']`$\neq$*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']`$\neq$*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 $P_1[3] =$ Š. Zato lahko trdimo, da vzorec P_1 je prisoten v besedi »KOKOŠ«. Za razliko od vzorca P_1 se za vrorec P_2 v vozlišču v preveri, ali velja v`.otroci['T']`$\neq$*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 ponovtiev 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 poizvedba *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 korenom 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 korenom 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 $O(n)$ notranjh 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 $T =$ »KOKOŠ«, za katero imamo zgrajeno priponsko drevo, ki je prikazano na Sliki 6. V besedi želimo preveriti, kolikokrat se pojavi vzorec $P =$ »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 $T[1:2] =$»KO«, in ker smo že preverili prvi znak, je potrebno preverit 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 korenom 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 $P =$ »KO« pojavi v besedi T dvakrat. Ča pa bi želeli poiskati indekse v besedi, kjer se vzorec pojavi, oziroma izvesti poizvedbo *seznamPojavov*(T, P), bi bil postopek enak. Pri tem bi beležili indekse pripon namesto štetja obiskanih listov. Ob obisku prvega lista bi se na konec praznega seznama vstavil indeks 1, za drugi list pa bi se na konec seznama vstavil še indeks 3. Torej bi poizvedba vrnila seznam $[1, 3]$.

3.2 GRADNJA

V tem podpoglavju bodo predstavljene različne metode 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 $O(n)$. Obstajata dve metodi izgradnje priponskega drevesa s časovno zahtevnostjo $O(n)$: McCreightov algoritem [4] in Ukkonenov algoritem [1]. V nadaljevanju bo bolj podrobno opisan Ukkonenov sprotni (angl. *on-line*) algoritem, saj v vsakem koraku podaljša vse pripone v drevesu za en znak. McCreightov algoritem pa ni sprotni algoritem, saj v i-tem koraku doda i-to najdaljšo pripono.

3.2.1 Naivna metoda

Naivna metoda je sprotna metoda gradnje priponskega drevesa. Ta metoda v vsakem koraku s sprehodom po drevesu podaljša vse pripone za en znak ter doda novo pripono. Metoda v i-tem koraku gradnje v dosedaj izgrajeno drevo, ki je bilo izgrajeno za podniz $T[1 : i-1]$, doda znak $T[i]$ ter tako izgradi drevo za podniz $T[1 : i]$. Naj bo α niz, ki z $x = T[i]$ tvori niz $T[k : i] = \alpha x$, pri čemer je $1 \leq k \leq i$ in posledično je α lahko tudi

prazen niz. Znak x je lahko dodan v drevo na tri načine:

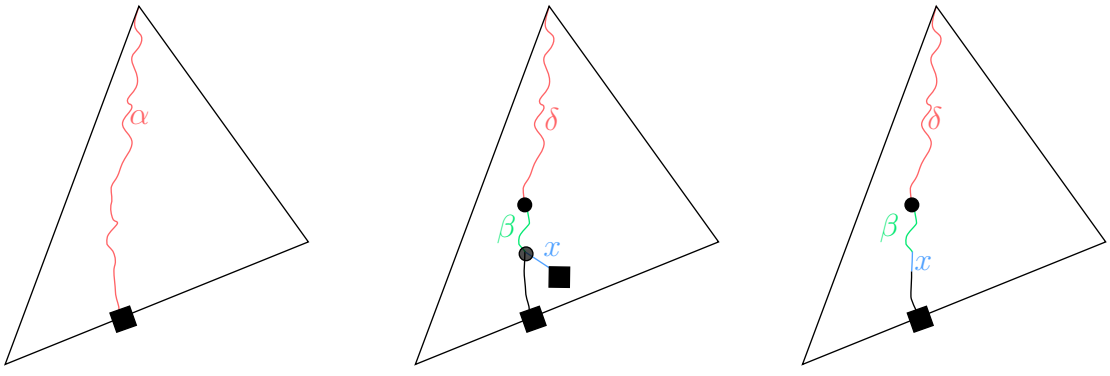
1. Če se niz α konča v listu, potem se zadnja povezava, ki je del niza α, podaljša za znak x. Torej enkrat, ko je list zgrajen, ne more postati notranje vozlišče. Način je prikazan na Sliki 7a.

2. Če pa se niz $\alpha = \delta\beta$ ne konča v listu, potem se konča bodisi v vozlišču v, bodisi na povezavi med vozliščema, recimo v_1 in v_2 ter δ predstavlja niz iz korena do vozlišča v_1, β pa predstavlja zadni del niza α 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 poveznava z oznako x. Če se niz α konča v vozlišču v, potem l postane otrok vozlišča v. Če pa se α 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 β. 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 7b.

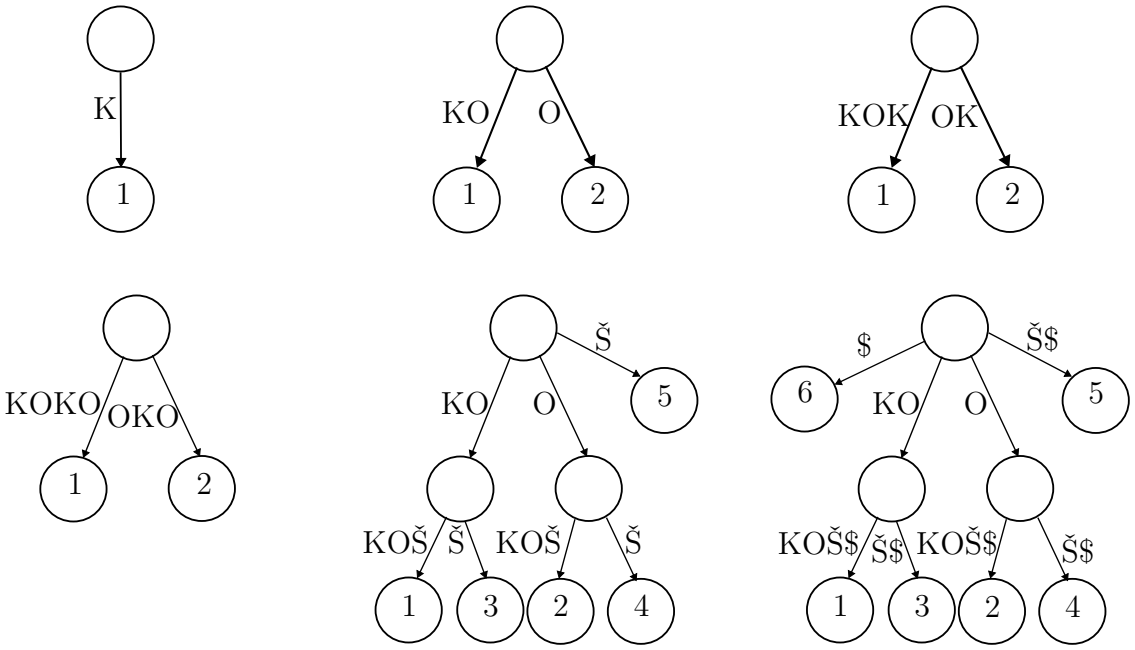
   (b) Če pa obstaja taka pot, ki se nadaljuje po nizu α z znakom x, se ne stori ničesar, saj je drevo že v implicitni obliki. Način je prikazan na Sliki 7c.

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



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

Opazimo, da drevo, ki je zgrajeno za podniz $T[1:i]$, $i < n$, ni nujno priponsko, saj niso vse pripone podniza $T[1:i]$ shranjene v listih. Še več, nekatere pripone niso niti eksplicitno predstavljene. Kljub temu pa to drevo vsebuje vso informacijo o pripadajočlh priponah, pri čemer moramo posebej beležiti vse implicitno predstavljene pripone.



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

Takšnemu drevesu rečemo implicitno priponsko drevo. Algoritem gradnje priponskega drevesa z naivno metodo je prikazan v Algoritmu 2. V algoritmu je uporabljena funkcija `najdiVozlišče`(α), ki vrne najglobje vozlišče pri iskanju niza α v drevesu. Primer gradnja priponskgea drevesa za besedo »KOKOŠ« z naivno metodo pa je prikazan na Sliki 8. Na Sliki 8 zgolj tretje (zadnje drevo v prvi vrstici) in četrto (prvo drevo v drugi vrstici) drevo sta implicitna priponska drevesa.

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

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

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

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

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

Algoritem 2: Naivna metoda gradnje priponskega drevesa

Vhod: Beseda T, dolžine n

Izhod: Priponsko drevo

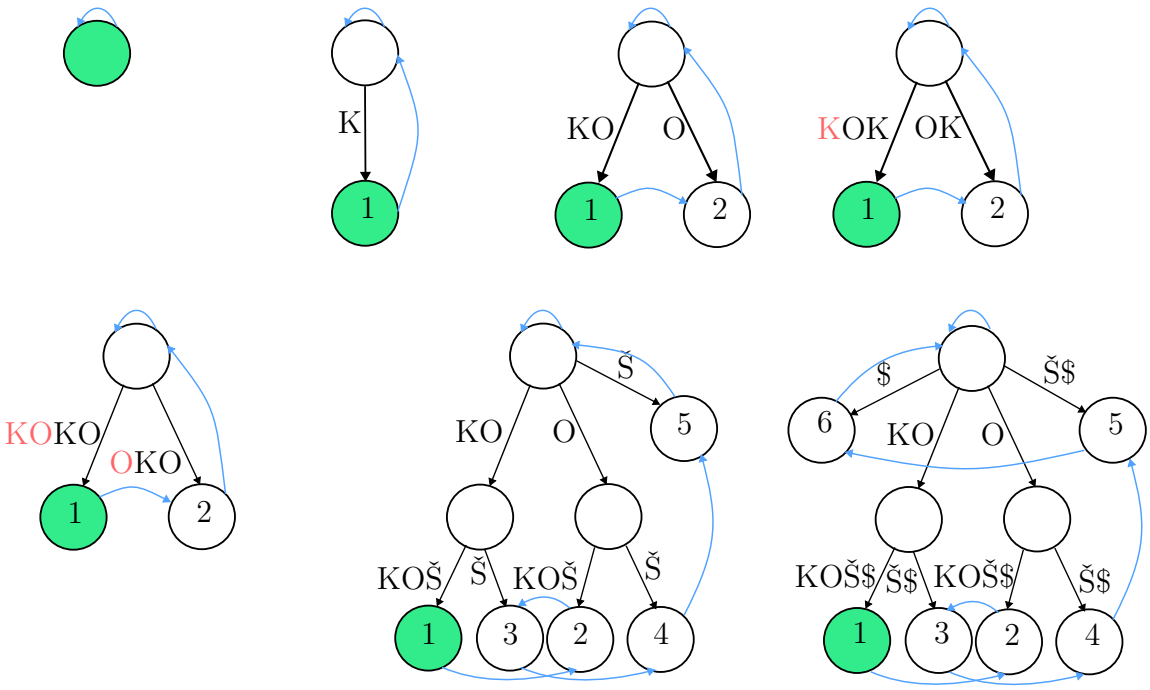
```
1  Ustvari vozlišče koren
2  skoren
3  za i = 1, ..., n
4      za j = 1, ..., i − 1
5          v ←najdiVozlišče(T[j, i])
6          u ← v.otrok[T[j + sd(v) + 1]]
7          če beseda(u)| = i − j potem
8              sicer če jeList(()u) potem
9                  Uporabi se način podalševanja pripon 1
10             sicer če u.otrok[T[i]] = NIL potem
11                 Ustvari list l_i, v.otrok[T[i]] ← l_i
12         sicer če | beseda(u)[i − j + 1] ≠ T[i] potem
13             Uporabi se način podalševanja pripon 2a
14     če koren.otrok[T[i]] = NIL potem
15         Ustvari list l_i, koren.otrok[T[i]] ← l_i
```

3.2.2 Izboljšana naivna metoda

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

Definicija 3.5. Povezava na naslednjo pripono Ψ iz vozlišča v, ki predstavlja pripono $T[i:]$, kaže na vozlišče w, ki predsatvlaj pripono $T[i+1:]$.

Na Sliki 6, ki prikazuje postopek izgradnje priponskega drevesa za besedo »KOKOŠ« z izboljšano naivno metodo, so prikazane povezave na naslednjo vozlišče z modro puščico. Uvedba seznama pripon omogoča izogib nepotrebnim sprehodom po drevesu. Tako metoda podaljša vse liste zgolj s sprehodom po seznamu. Pritem velja, da je $\Psi(koren()) = koren()$. Na ta način lahko pregledamo vse pripone v drevesu. Me-



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

toda se v vsakem koraku sprehodi iz začetne točke, ki predstavlja najdaljšo pripono, po seznamu pripon. Na Sliki 9 je začetna točka označena z zeleno barvo. Ker je število vozliš v seznamu lahkomanjše od števila pripon in *koren*() kaže nase, metoda šteje koliko pripon je že podaljšala v trenutnem koraku. Na Sliki 6 so vse implcitno predstavljene pripone, prikazane z rdečo barvo.

V Algoritmu 3 je prikazana psevdokodo izbolšane naivne metode gradnje. V vsakem koraku izgradnje se metoda sprehodi po seznamu vozlišč, ki predstavljajo pripono, in podaljša vse pripone. Vsakič, ko se pripona podaljša z načinom 2a, je potrebno poraviti seznam vozlišč. V vsakem koraku je v seznamu največ toliko vozliš kot je pripon. Ker je lahko vozlišč manj, metoda šteje število že obiskanih pripon in vse ne obiskane pripone se obišče iz korena.

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

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

Algoritem 3: Izboljšana naivna metoda gradnje priponskega drevesa

Vhod: Beseda T, dolžine n

Izhod: Priponsko drevo

```
1  Ustvari vozlišče koren; ZacetnaTocka ← koren()
2  za i = 1, ..., n
3      v ← ZacetnaTocka
4      za j = 1, ..., i − 1
5          če jeList(v) potem  Uporabi se način podalševanja pripon 1
6          sicer
7              k ← |beseda(v)|, u ← v.otrok[T[j + k + 1]]
8              če u = NIL potem
9                  Ustvari list l_i; v.otrok[T[i]] ← l_i
10                 Popravi seznam pripon (v spremni v l_i)
11             sicer če j + |beseda(u)| < i potem
12                 če u.otrok[T[i]] = NIL potem
13                     Ustvari list l_i, u.otrok[T[i]] ← l_i
14                     Popravi seznam pripon (v spremni v l_i)
15                 sicer  Popravi seznam pripon (v spremni v u)
16             sicer če |beseda(u)[i − j + 1] ≠ T[i] potem
17                 Uporabi se način podalševanja pripon 2a
18                 Popravi seznam pripon (v spremni v l_i)
19         v ← v.Ψ
20     če koren.otrok[T[i]] = NIL potem Ustvari list l_i; koren.otrok[T[i]] ← l_i
21     če ZacetnaTocka = koren() potem ZacetnaTocka ← l_1
```

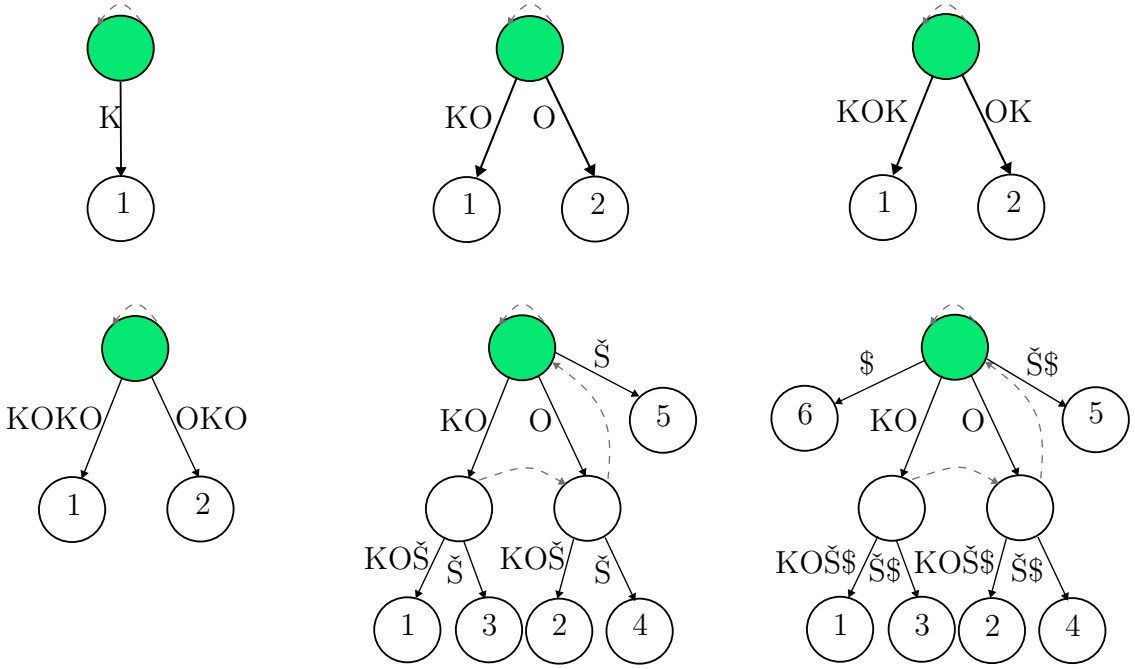
Čeprav te izboljšave znižajo čas izgradnje priponskega drevesa iz $O(n^3)$ na $O(n^2)$, ampak taka izboljšava ne omogoča izgradnje drevesa v času $O(n)$. Za izgradnjo priponskega drevesa v linearnem času obstajata dva algoritma: McCreightov algoritem [4] in Ukkonenov algoritem [1].

3.2.3 Ukkonenov algoritem

Ukkonenov algoritem deluje na podoben način naivna metoda in izboljšana 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 obstoječe drevo črko $T[i]$. Črka je dodana v drevo na iste 3 načine kot pri ostalih dveh metodah in jih lahko vidimo tudi na primeru izgradnje

priponskega drevesa za besedo »KOKOŠ$«. Postopek izgranje z Ukkonenovim algoritmom in vmesna drevesa so prikazana na Sliki 10. Iz načina dodajanja novih znakov v priponsko drevo, ki ga uporablja algoritem, se opazi, da se število vozlišč v drevesu spremeni samo z drugim načinom dodajanja (2a). Torej se moramo sprehoditi zgolj po vozliščih, iz katerih kažejo povezave s konci implicitnih pripon. V teh točkah se lahko zgodi delitev povezave. Zato se potrebuje dodatno povezavo, ki bo nadomestila povezavo na naslednjo pripono, imenovano priponska povezava (angl. *Suffix link*).

Definicija 3.7. Vozlišče v je notranje vozlišče in $beseda(v) = x \cdot \alpha$. Priponska povezava $sl(v)$ je povezava na notranje vozlišče w in $beseda(w) = \alpha$.



Slika 10: Primer izgradnje priponskega drevesa z uporabo Ukkonenovaga algoritma za besedo »KOKOŠ$«.

Podobno kot pri povezavi na naslednjo pripono, tudi $sl(koren()) = koren()$. Na Sliki 10 so priponske povezave označene z sivo črtkano črto. Vse točke, v katerih se bo v tem koraku zgodila delitev, 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 10 je z zeleno označeno vozlišče, ki 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. Zato 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 listi drevesa, saj od takrat ni več potrebno v trenutnem koraku izgradnje deliti povezave in ustavrjati nova vozlišča. Pri tem velja tudi, da končna točka v koraku $i-1$ postane začetna aktivna točka v koraku i, saj se prva nova delitev v koraku i lahko zgodi zgolj v točki, kjer so se končale delitve v koraku $i-1$, in to je končna točka koraka $i-1$.

Vsako aktivno točko lahko predstavimo kot referenčni par (s, α), pri čemer je s vozlišče pred aktivno točko in α je niz iz vozlišča s do aktivne točke. Za lažje shranjevanje niza α 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 najglobje vozlišče pred aktivno točko, takšen par imenujemo kanonična oblika referenčenga para. Kanonična oblika je po definiciji enolična. Med vsemi referenčnimi pari za dano aktivno točko velja, da je pri paru, ki je v kanonični obliki, niz $\alpha = T[k : p]$ najkrajši. Na Sliki 10 v četrtem koraku 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 vozlišče. 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 vozlišče 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 4. V algoritmu 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 katerga je bil nazadnje pripet list v i-tem koraku. Vozlišče $sVoz$ je *NIL* zgolj v prvem dodajanju novega lista v vsakem koraku.

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

Funkcija `razdeliTestiraj` prejme kot vhod trenutno aktivno točko, ki je podana kot referenčni par $(s, (k, p))$ v kanonični obliki, ter znak t, ki se želi vstaviti v priponsko drevo. Funkcija preveri, ali se niza $T[k : p+1]$ in $T[k : p]{\cdot}t$ ujemata. Če se 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 4: Ukkonenov algoritem za izgradnjo priponskega drevesa

Vhod: Beseda T, dolžine n

Izhod: Priponsko drevo

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

Izrek 3.8. *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 seštevalne 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` obi-

šče $O(n)$ 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 β z vsakim obiskanim vozliščem skrajša, saj se poveča število k. Niz β pa se lahko poveča zgolj v 15 vrstici Algoritma 4. Ker se niz β poveča n-krat skozi celotno izgradnjo, potemtakem se tudi niz β 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 4 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.

□

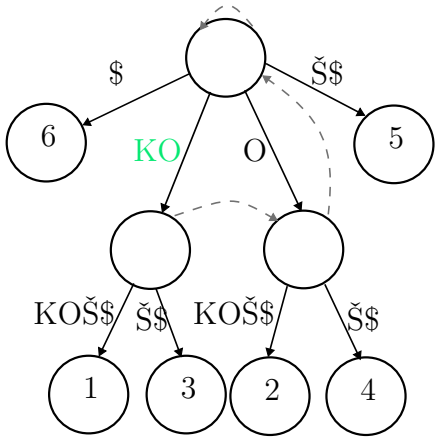
Zaključek. Tako McCreight algoritem [4] kot tudi Ukkonenov algoritem [1] zgradita priponsko drevo v času $O(n)$. Oba algoritma izdelata priponsko drevo, ki ima enake priponske povezave med vozlišči, pri tem pa se zalikujeta v umesnih drevesih. V McCreightovem algoritmu vmesno drevo v i-tem koraku predstavlja 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 Σ 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 6.2, kjer bo tudi izmerjen vpliv pomanjkanja notranjega pomnilnika ter posledična uporaba zunanjega pomnilnika (`Swap` razdelek na zunanjem spominu) namesto notranjega pomnilnika.

4 PRIPONSKO POLJE

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



	1	2	3	4	5	6
LCP	-	0	2	0	1	0
SA	6	1	3	2	4	5

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

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

$2(v + l - 1)$ celih števil oziroma med $2n$ in $4n - 2$ celih števil.

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

Pri tem pa se pojavi vprašanje, ali je mogoče indeksirati celotno besedo T z zgolj n-timi celimi števili. Odgovor je da, saj vsak list v drevesu predstavlja eno pripono. To lahko storimo tako, da shranimo indekse, ki so shranjeni v listih priponskega drevesa, v polju celih števil in sicer polnemo polje iz leve proti desni z indeksi, ki jih dobimo ob premem sprehodu po drevesu. Pri tem pa se pojavi problem iskanja po takem polju, saj nam premi sprehod ne zagotavlja leksikografske urejenosti pripon. Brez škode za splošnost lahko predposvaimo, da so povezave v priponskem drevesu leksikografsko urejene, torej velja $beseda(l_i) < beseda(l_{i+1})$, kot je bilo to storjeno na vseh slikah do zdaj. Torej imamo polje indeksov pripon, ki so leksigrafsko urejeni, in tako polje imenuje priponsko polje (angl. *Suffix array* oziroma SA). Primer priponskega polja za besedo »KOKOŠ$« je prikazan na spodnjem delu Slike 11. Poleg priponskega polja označenega SA je na sliki prikazno tudi LCP polje.

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

4.1 POIZVEDBE

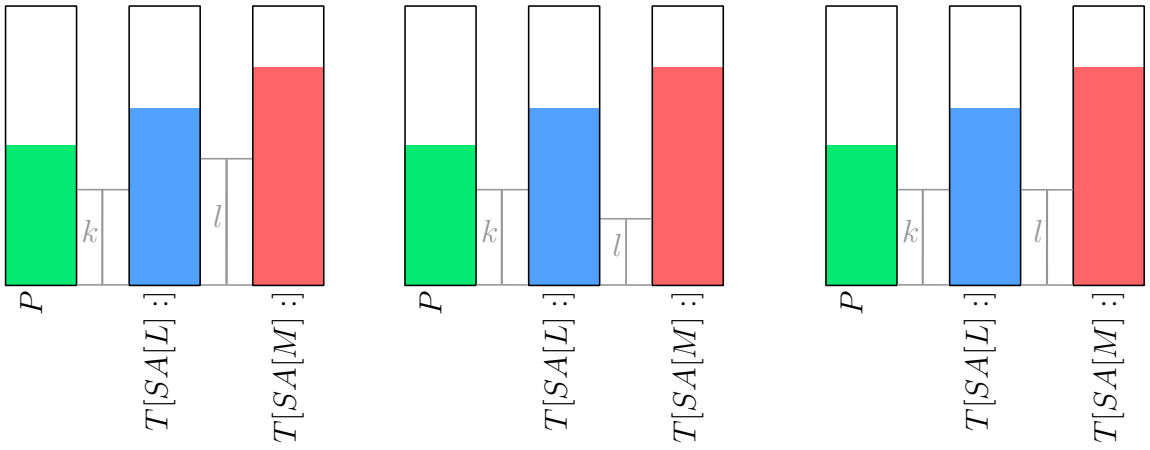
Priponsko polje je bilo izgrajeno kot alternativa priponskemu drevesu, ki potrebuje manj prostora. Zato se ga uporablja za indeksiranje besede T in posledično iskanje vzorcev v njej. V tem podpoglavju bodo predstavljene implementacije za priponsko polje istih poizvedb, ki so bile predstavljene za priponsko 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 razpolavljanjem, saj so pripone v priponskem polju urejene. V vsakem koraku razpolavljanja 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 razpolavljanje potrebuje $O(\log n)$ primerjav, vsaka primerjava pa potrebuje dodatnih $O(m)$ primerjav, ali se pripona in vzorec ujemata. Ta način iskanja je $O(\log n)$-krat počasnejši od iskanja v priponskem drevesu. To razliko si želimo znižati, pri tem pa ne želimo shraniti celotne topologije drevesa, ampak zgolj informacije, ki pospešijo iskanje po priponskem polju. Informacijo, ki jo želimo shraniti, je število znakov, ki se ujemajo med dvema priponama, ali z drugimi besedami dolžino najdaljše predpone. To informacijo shanimo v polje najdaljših skupnih predpon (angl. *Longest common prefix* oziroma LCP), ki za vsak pari indeksov i, j hrani

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

Tako definirano LCP polje je potrebno $O(n^2)$ prostora. Problem, katerega si želimo znebiti z LCP poljem, je ponovnega štetja, znakov, ki vemo, da se ujemajo med pripono $T[SA[M] :]$ in P. Le teh je v vsakem koraku k. S tem vedenjem lahko zmanjšamo število primerjav črk med P in priponami $T[SA[M] :]$ na $O(m)$ skozi celotno izvajanje razpolavljanja.



(a) $k < l = LCP[L, M]$ (b) $k > l = LCP[L, M]$ (c) $k = l = LCP[L, M]$

Slika 12: Prikaz razmerja med P in priponsama

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 $LCP[L, M]$-tih

znakih ter pripona $SA[L]$ je leksikografsko manjša od $SA[M]$. Torej obstajajo tri možne relacije med k in $LCP[L, M]$:

1. $k < LCP[L, M]$, potem se $SA[L]$-ta pripona in $SA[M]$-ta pripona bolj ujemata kot $SA[L]$-ta pripona in P. Ker smo v predhodnem koraku izvedeli, da je P večji od 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 $SA[M : R]$.

2. $k > LCP[L, M]$, potem se $SA[L]$-ta pripona in $SA[M]$-ta pripona manj ujemata kot $SA[L]$-ta pripona in P in označimo $LCP[L, M] = l$. Torej $P[l+1] = T[SA[L+l+1]] < T[SA[M+l+1]]$, saj je $L < M$, ker je $SA[L]$-ta pripona leksikografsko manjša $SA[M]$-ta pripona. Torej se bisekcija nadaljuje na intervalu $SA[L : M]$.

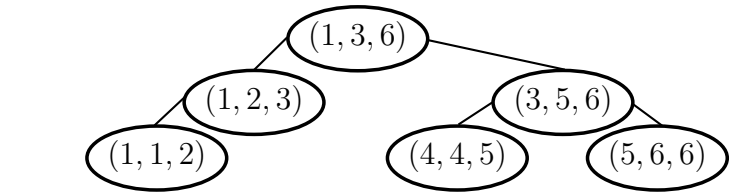
3. $k = LCP[L, M]$, potem je potrebno preveriti, ali je P večji od $SA[M]$-te pripone. Pri tem ni potrebno preveriti prvih k znakov, saj vemo, da se ujemajo, ker je se $SA[L]$-ta pripona ujema s $SA[M]$-ta pripona v k znakih in se $SA[L]$-ta pripona tudi ujema z P v k znakih. Med preverjanjem štejemo, 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 $SA[L : M]$, sicer je $P[k + k' + 1] > T[SA[M] + k + k' + 1]$ in nadaljujemo v interval $SA[M : R]$. Preden nadaljujemo v naslednji interval popravimo $k \leftarrow k + k'$, pri čemer novi $k \leq m$.

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

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

Pri tem se opazi, da velika večina celic LCP polja ne bo nikoli uporabljena pri bisekciji. V vsakem koraku bisekcije se preverja, ali je vzorec večji od sredinske točke M na intervalu $L : R$ indeksov v priponskem polju. 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 13, na kateri je prikazano tudi drevo sledi bisekcije. Na vsakem vozlišču drevesa je prikazan tudi interval v indeksov priponskega polja (L, M, R), pričemer L predstavlja začetek intervala, Rpredstavlja konec intervala indeksev in M predstavlja sredinski indeks, katerega predstavljena pripona bo bila primerjana z vzorcem P [24].

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



	1	2	3	4	5	6
SA	6	1	3	2	4	5
Llcp	1	0	0	0	0	0
Rlcp	0	2	0	1	0	2

Slika 13: Primer *L-LCP* in *R-LCP* polji za priponskega polja nad besedo »KOKOŠ$«.

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

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

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

Idejo poizvedbe *prisotnost*(T, P) lahko uporabimo za implementacijo poizvedbe *številoPonovitev*(T, P). Poizvedba vrne število *occ*, ki je število ponovite vzorca P v besedi T. Naivna implementacija bi bila štetje pripon levo in desno od M-te pripone, ki je prva pripona v bisekciji, ki se ujema z P. Ta postopek potrebuje $O(m + \log n + occ)$ časa. Poizvedbo se lahko pohitri na $O(m + \log n)$ z dvema 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 $SA[L : R]$, ki se začnejo s P, za izdelati seznam indeksov ponovitev vzorca P v besedi T. Za najti interval $SA[S : E]$ 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.

4.2 GRADNJA

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 zgraditi priponsko polje. Pri tem pa ponovno predpostavimo, da so listi v drevesu leksikografsko urejeni. Priponsko polje je zgrajeno v dveh korakih:

1. izgradnja priponskega drevesa z $O(n)$ algoritmom, recimo z Ukkonenovim algoritmom (korak ni potreben če, je priponsko drevo že 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 zgradi priponsko polje in LCP polje v $O(n)$ časa. Vsak korak potrebuje $O(n)$ časa, saj smo izbrali algoritem za izgradnjo priponskega drevesa s časovno zahtevnostjo $O(n)$ in sprehod po drevesu z $O(n)$ vozlišči potrebuje $O(n)$ časa. Pri tem pa algoritem potrebuje doatni prostor za največ $10n - 7$ števil. Razlog za izgradnjo priponskega polja je zmanjšanje količine spomina, ki ga zasede indeks besede. Zato bi potrebovali algoritem, ki zgradi priponsko polje v $O(n)$ časa in ne 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 (*Quck sort* oziroma *Merge sort*) potrebujejo $O(n \log n)$ primerjav za izgradnjo priponskega polja. Ker so pripone urejene leksikografsko, vsaka primerjava potrebuje $O(n)$ časa, torej celotna izgradnja potrebuje $O(n^2 \log n)$ časa. Pri tem se potrebuje še dodatnega $O(n)$ časa za izgradnjo LCP polja, saj je potrebno izračunati vse *lcp* vrednosti zaporednih pripon [14].

Namesto tega lahko uporabimo korensko urejanje (angl. *RadixSort*). Torej v i-tem koraku korenskega urejanja so pripone urejene v vedrih glede na prvih i znakov.

To dejstvo se lahko uporabi za izgradnjo LCP polja, saj se v i-tem koraku lahko zapiše vrednost i na začetek vsakega na novo ustvarjenega vedra. Korensko urejanje potrebuje $O(kn)$ časa, pri čemer je k dolžina korena, in v najslabšem primeru je $k = n$, torej metoda potrebuje $O(n^2)$ časa.

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

Obstaja pa bolj učinkovit algoritem za zgraditi priponskega pola, ki so ga predlagala Ko in Aluru [25] in potrebuje $O(n)$ časa. Ideja algoritma temelji na deljenju pripon na L pripone, za katere velja, da je $T[i:]$ je manjša od $T[i+1:]$, in S pripone, za katere pa velja, da je $T[i:]$ večja $T[i+1:]$, 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:

1. deljenje pripon na S in L,
2. ureditev polja glede na prvi znak pripone,
3. premik L pripon na začetek veder s sprotnim urejanjem S pripon,
4. 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 poljaštevil dolžine n in 3 dodatna bitna polja (dva dolžine n in enega dolžine $n/2$) [25].

4.3 SIMULACIJA OPERACIJ PRIPONSKEGA DREVESA

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

Če namesto *Q-LCP* polja, vzamemo vrednost v *L-LCP*, in sicer vrednost *L-LCP*$[M]$ predstavlja dolžno najdaljše skupne predpone $lcp(T[SA[M]\ :], T[SA[L]\ :])$, pri čemer je L začetni indeks intervala bisekcije s sredinsko točko v M, in vrednost funkcije lcp označimo kot k. Vemo tudi, da je $SA[L]$ leksikografsko manjši od $SA[M]$, 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 $k \leq lcp(T[SA[i-1]\ :], T[SA[i]\ :])$. Posledično obstaja tak i, za katerega velja $lcp(T[SA[i-1]\ :], T[SA[i]\ :]) = k$. Podoben sklep se lahko naredi tudi za *R-LCP*$[M]$, pri čemer uporabimo interval v priponskem polju med M in R. Zato se lahko naredi bolj splošno LCP polje $LCP[2:n]$, ki ima verdnosti

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

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

Novo LCP polje potrebuje zgolj $n-1$ celih števil. Pri tem pa potrebuje dodatno podatkovno strukturo za učinkovito iskanje najmanjše vrednosti na intervalu oziroma *rmq*, ki je potrebna za nadomestiti *Q-LCP* polje. Prva možnost je izgradnja *rmM*-drevesa, ki v vsakem vozlišču vsebuje zgolj vrednost m. Iskanje v drevesu pa potrebuje $O(\log n)$ časa oziroma $O(m \log n + \log n)$ časa za poizvedbo, kar pa je prepočasno, saj potrebuje iskanje z *Q-LCP* poljem $O(m + \log n)$ 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 $v = lca(l_{i-1}, l_i)$, pričemer 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 korenom v v. Ker je priponsko polje ekvivalentno listom priponskega drevesa, so vsi listi poddrevesa s korenom 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 z definicijo 4.4 [26].

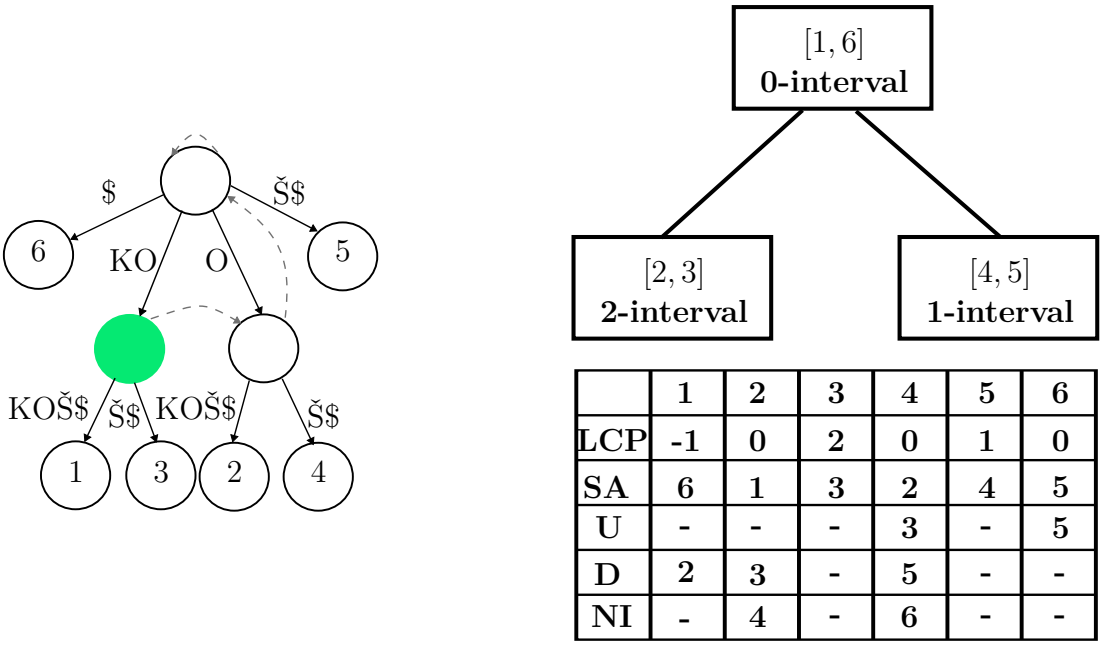
Definicija 4.4. Interval $[i:j]$ je l-interval v LCP polju, natanko tedaj ko velja:

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

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

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

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



	1	2	3	4	5	6
LCP	**-1**	**0**	**2**	**0**	**1**	**0**
SA	**6**	**1**	**3**	**2**	**4**	**5**
U	**-**	**-**	**-**	**3**	**-**	**5**
D	**2**	**3**	**-**	**5**	**-**	**-**
NI	**-**	**4**	**-**	**6**	**-**	**-**

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

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

Opazi se lahko, da L od naslednjega brata od v je $R + 1$ od v oziroma je enaka vrednosti R od starša od v. Torej vrednosti R ni potrebo hraniti, saj se jo lahko naračuna. Vrednost l se lahko izračuna s pomočjo LCP polja, kot

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

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

Na ta način lahko vrednostiv D, NI in U spremenimo v poja dolžine n in jih

poravnamo z začetki intervala L. Toren je vrednost $D[L]$ enaka začetku intervala naslednjega intervala od najdaljšega intervala, ki se začne z indeksom L. Vrednost $D[i]$ 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 $LCP[j]$. Vrednost $NI[i]$ 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 $LCP[j]$. Vrednost $U[i]$ pa hrani položaj prvega indeksa j manjšega od i, za katerega velja $LCP[j] > LCP[i]$ in vse vrednosti v LCP med indeksi j in i so večje ali enake od $LCP[j]$. Torej za interval $SA[i : j]$ velja $D[i] = U[j + 1]$. Začetek intervala drugega otroka v prvem podintevalu intervala $SA[i : j]$ izračunamo kot $U[D[i]]$. S temi polji se lahko sprehodimo po priponskem polju, kot bi se sprehodili po priponskem drevesu, pri tem pa potrebujemo $5n$ števil za shraniti vse te vrednosti [26]. Na Sliki 14 je prikazan tudi zapis intervalnega drevesa s polji U, D in NI.

Poizvedbe. Poizvedba se začne z indeksom $s = 1$ in $e = n$ in preverili smo prvih $o = 0$ znakov vzorca P. Nato povečamo vrednost o za ena in najdemo interval, ki se začne z vrednostjo $P[o]$, z Algoritmom 5. Nato v vsakem koraku iskanja za trenutni interval $[s : e]$ poiščemo vrednost $l = \min_{s<k\leq e} LCP[k]$. Vemo tudi, da se prvih o znakov P-ja ujema s priponami na intervalu $[s : e]$. Nato preverimo, ali je $P[o : k] = T[SA[s] + o : SA[s] + k]$, pri čemer je $k = \min\{l, m\}$. Če se ujemata, popravimo $o \leftarrow k$ in začnemo iskati podinterval $[s' : e']$, za katerega velja $P[o+1] = T[SA[s']+o+1]$, ki bo postal novi interval $[s : e]$. Če je $o = m$, lahko vrnemo, da je P prisoten v T, poročamo, da se P ponovi $(s-e+1)$-krat v T, ali pa izgradimo seznam indeksov besede $[SA[s], \ldots, SA[e]]$, odvisno od poizvedbe. Če pa se $P[o : k]$ ne ujema s $T[SA[s]+o : SA[s]+k]$ pa vrnemo, da P ni prisoten v T, 0 ali prazen seznam, odvisno od poizvedbe. Iskanje podintervala je implementirano s polji D in U za najti prvi podinterval ter s poljem NI za najti interval, ki se začne z znakom $P[o + 1]$, 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 $[s' : e']$ prikazuje Algoritm 5 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 operacij nad priponskim drevesom z uporabo priponskega polja in LCP polja ter l-intervalov potrebuje $O(m)$ časa za poizvedbo *prisotnost*(T, P) ter omogoča pospešitev časa poizvedbe *številoPonovitev*(T, P) na $O(m)$. V priponskem drevesu ta poizvedba potrebuje $O(m+occ)$ časa, saj je potrebno prešteti število listov v poddrevesu. Z uporabo simulacije priponskega drevesa tega štetja ni potrebno storiti, saj poznamo velikost intervala, ki ga pokriva vozlišče, ki je koren poddrevesa, čigar pripone se začnejo z iskanim vzorcem. Število pripon je natanko razlika med začetkom in koncem interval, pri čemer konec intervala je začetek naslednjega intervala, ki je

Algoritem 5: Algoritem za iskanje pod intervala

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

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

```
1  če s < U[e + 1] ≤ e potem
2      s' ← U[e + 1]
3  sicer
4      s' ← D[s]
5  če T[SA[s] + l + 1] == c potem
6      vrni [s, s' − 1]
7  dokler NI[s'] ≠ −1
8      e' ← NI[s']
9      če T[SA[s'] + l + 1] == c potem
10         vrni [s', e' − 1]
11     s' ← e'
12 če T[SA[s'] + l + 1] == c potem
13     vrni [s', e]
14 vrni [−1, −1]
```

shranjen v polju NI. Poizvedba *seznamPojavov*(T, P) še vedno potrebuje $O(m + occ)$ časa, ker je potrebno prehoditi priponsko pole.

Opomba. Metoda iskanja z LCP intervali se lahko uporabi tudi za iskanje po ostalih kompaktnih številskih drevesih (na primer Patricijinih drevesih). Pri tem se nadomesti priponsko polje s poljem listov LA (angl. *leaf array*), kjer i-ta celica polja predstavlja besedo, ki jo predstavlja i-ti list številskega drevesa. Pri tem pa ostaja definicija LCP polja podobna, in sicer je definirano tako, da so za vsaki i med 2 in n vrednosti $LCP[i] = lcp(LA[i-1], LA[i])$ in vrednost $LCP[1] = -1$.

5 KOMPAKTNA PREDSTAVITEV PRIPONSKEGA DREVESA

Za daljše besede T lahko priponska drevesa presežejo velikost notranjega pomnilnika. Prva možna rešitev tega problema je uporaba priponskega polja ter LCP polja za simulirati operacije nad priponskim drevesom, ampak ta rešitev za dvakrat daljše besede se ponovno pojavi ta problem. Torej potrebujemo rešitev, ki bo omogočala še bolj prostorsko učinkovito rešitev.

Vsako vozlišče priponskega drevesa hrani niz, ki prestavlja 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 vštet v abecedo tudi, znak za konec besedila označen kot »$«), potrebuje med 60 in 144 GB notranjega pomnilnik. 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 [13].

Ta problem se da rešiti s kompaktno predstavitvijo podatkovnih struktur. Podatkovna struktura, ki bo predstavljena 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. Večina potrebnih operacije so operacije nad drevesi, ki so bile predstavljene na Definiciji 2.3. Pri tem so dodane tudi specifične operacije, ki so potrebne za pravilno delovanje priponskega drevesa, ter nekatere operacije so popravljene za delovanje z znaki iz abecede Σ. Operacije, ki jih podpira priponsko drevo, so predstavljene na Definiciji 5.1 [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. $sd(v)$ 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 4.3 je bil predstavljen način simulacije priponskega drevesa s pomočjo priponskega polja SA in polja najdaljših skupnih predpon LCP. Torej ideja kompaktne predstavitve priponskega drevesa bo tudi temeljila na teh dveh podatkovnih strukturah. Pri tem pa bomo uporabili, kompaktni različici podatkovnih struktur. 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. Iz tega sledi definicija za podatkovno strukturo kompaktno priponsko drevo.

Definicija 5.2. Kompaktno priponsko drevo nad besedilom T je sestavljeno iz:

1. kompaktne predstavitve topologije drevesa τ,
2. kompaktnega zapisa priponskega polja SA,
3. kompaktne predstavitev polja najdaljših skupnih predpon LCP.

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

Topologija drevesa. Ker se za kompaktno priponsko drevo uporablja BP kot predstavitev topologije drevesa, je potrebno podpirati operacije $rang_p$, $izbira_p$, *zapri*, *odpri* in *oklepa*, ki se izvajajo v konstantnem času za $p \in \{0,1\}^*$. Pri tem so potrebni dve dodatni podatkovni strukturi za operacijo *rang* ($rang_0$, $rang_{01}$) in tri za operacijo *izbira* ($izbira_0$, $izbira_1$ in $izbira_{01}$). Operacije *zapri*, *odpri* in *oklepa* pa uporabljajo podatkovno strukturo za višek (razlika med uklepaji in zaklepaji), ki sta jo predlagala Munro in Raman [15], implementirano na podoben način kot dodatna podatkovna struktura za *rang*, ki nadomesti rmM-drevesa. To podatkovno strukturo imenujemo L, ki hrani polje najnižjih viškov L' in v celici $L'[i]$ 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 tabelov in vrednost $M[i][k]$ hrani 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 zadnje 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 $rMq(\cdot,,)$ operaciji, ali $|\Sigma| > \log n$, se lahko doda še rmM-drevo (angl *range minumum maximum tree*, predstavljeno v podpoglavju 2.4), 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 τ nad besedilom T dolžine n potrebuje največ $4n + o(n)$ bitov.*

Dokaz. Priponsko drevo nad besedilom T ima največ $2n-1$ vozlišč: od teh je $n-1$ notranjih vozlišč in n listov. Vsako vozlišče se predstavi kot par oklepajev, torej potrebujemo 2 bita za predstaviti vsako vozlišče. Torej je potrebnih največ $4n-2$ bitov za zapis topologije drevesa τ s sekvenco uravnoteženih oklepajev. Zato celotna podatkovna struktura topologije drevesa potrebuje največ $4n + o(n)$ bitov, saj vsaka dodatna podatkovna struktura potrebuje še dodatnih $o(n)$ bitov. □

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 Ψ, druga implementacija imenovana FM-indeks pa uporablja LF funkcijo, ki je inverzna funkcija od Ψ. Funkcija $\Psi(i)$ vrne položaj pripone $i + 1$ pripone v priponskem polju ali z drugimi besedami

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

pri čemer je $SA^{-1}[i]$ hrani polažaj pripone $T[i :]$ v priponskem polju SA in polje $SA^{-1}[1 : n]$ imenujemo inverzno priponsko polje. Torej s funkcijo Ψ se lahko sprehajamo po priponskem polju oziroma besedi T iz leve prozi desni. Funkcija LF pa omogoča sprehod vobratni smeri, iz desne proti levi [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 v kompaktnem priponskem drevesu, podpira sledeče operacije, ki potrebujeo ča:

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

Funkcija Ψ je uporabna v kompaktnih priponskih drevesih za izgradnjo priponskih povezav v času t_Ψ, čeprav ni potrebna za pravilno delovanje priponskega polja. Iz definicije funkcije Ψ je razvidno, da jo lahko simuliramo z uporabo operacij $inverz(\cdot)$ in $pripona(\cdot)$. Zato je čas potreben za izvršit funkcijo Ψ $t_\Psi \leq t_{SA}$, ki ga potrebujeta operaci $inverz(\cdot)$ in $pripona(\cdot)$. Iz Definicije 5.4 sledi, da mora kompaktno priponsko polje podpirati funkcijo Ψ, zato implementacije s FM-indeksom ne pridejo v poštev, saj morajo funkcijo Ψ simulirati kot $inverz(pripona(\cdot))$.

V nadaljevanju tega dela poglavja bo predstavljena preprosta različica kopaktenga priponskega polja. Ta različica je bila izbrana zaradi enostavne implementacije operaciji ter preproste vizualizacije podatkovne strukture. V tej implementacije so vse štiri operacije implementirane z uporabo Ψ 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 Ψ je predstavljena z istoimenskim poljem, katerega i-ta celica je $\Psi[i] = \Psi(i) = SA^{-1}[SA[i]+1]$. Ideja kompaktnega zapisa priponskega polja temelji na dejstvu, da se lahko premikamo po besedilu in posledično med priponami zgolj z uporabo Ψ funkcije, saj je $SA[\Psi[i]] = SA[i]+1$. Torej se lahko znebimo besede T in jo zamenjamo z bitno polje spremeb 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 Ψ razdeli polje Ψ na $|\Sigma|$ delov in uvede polje C začetkov intervalov v Ψ polju, ki se začnejo s poljubnim znakom $c \in \Sigma$, oziroma $C[c] = i$ za poljuben znak iz Σ, 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 Ψ polje s polji Ψ_c; $c \in \Sigma$, kjer je $\Psi[i] = \Psi_c[i']$, za $i' = i - C[S[rang_1(D, i)]]$. Vsako polje Ψ_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 ohranimo možnost opravljanja operacije $izbira_1$ v konstantnem času, kar pomeni, da funkcija Ψ 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 s priponami z istim začetnim zankom, pri tem pa ne lokacije pojavov vzorca v besedilu. Za to sta potrebni dodatni podatkovni strukturi, ki nadomestita priponsko polje SA ter inverzno polje SA^{-1}. Polji se lahko nadomestita s poljema vzorcev. Priponsko polje SA se vzorči $l = \Theta(\log n)$-krat, kar je dober kompromis med časovno in prostorsko zahtevnostjo. Pri tem se uporabi dodatno bitno polje $B[1, n]$, kjer $B[i] = 1$ natanko tedaj, ko $i = 1$ ali $SA[i] \mod l = 0$. 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 Ψ 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 polje vzorči SA^{-1} na enakomernih intervalih dolžine $l = \Theta(\log n)$, kar je dober kompromis med časovno in prostorsko zahtevnostjo. Torej ima i-ta celica polja $SA_S^{-1}[1, \lfloor n/l \rfloor]$ vrednost $SA_S^{-1}[i] = SA^{-1}[il]$. Poljubno vrednost SA^{-1} se izračuna tako, da se najprej izračuna $i' = \lfloor i/l \rfloor l$, nato se $i - i'$-krat aplicira funkcija Ψ nad vrednostjo $j' = SA_S^{-1}[i'/l]$ oziroma $SA^{-1}[i] = \Psi^{i-i'}[j'] = \Psi^{i-i'}[SA_S^{-1}[i'/l]]$. Isti razmislek kot za časovno zahtevnost dostopa do polja SA velja tudi za časovno zahtevnost dostopa do polja SA^{-1}, ki ima časovno zahtevnost dostopa $t_{SA} = O(l) = O(\log n)$. Pri tem velja omeniti, da če se inverzno priponsko polje

uporablja bolj poredko, se lahko vzorči z večjim faktorjem l kot se je vzorčilo priponsko polje SA. Na ta način se zmanjša prostorsko zahtevnost, ampak se poslabšajo časovne zahtevnosti operacije, ki se jo bolj poredko uporablja [6].

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

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

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

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 priponi $T[j :]$ in $T[i :]$, kjer je $T[i :] < T[j :]$ in imata najdaljšo skupno predpono dolžine $PLCP[j] > 0$. Potem ima $T[i + 1 :]$ in $T[j + 1 :]$ najdaljšo skupno predpono dolžine $PLCP[j]-1$. Torej je $PLCP[j+1] = PLCP[j]-1$. Posledično velja, da je $PLCP[j]+2j < PLCP[j+1]+2(j+1) = PLCP[j]+2j+1$. Ker je $PLCP[n-1]+2(n-1) < 2n$, saj ima lahko $T[n-1 :]$ dolžino najdaljše skupne predpone s poljubno pripono 1. Zato 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 $izbiro_1$ [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 kompaktnega 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.*

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. □

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

Dokaz. Ker obstajajo različne implementacije kompaktnega 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 τ. Iz Leme 5.5 pa sledi, da je potrebnih $2n + o(n)$ bitov za shraniti polje LCP.

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

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

Posledica 5.7. *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 $O(\epsilon^{-1} n \log |\Sigma|)$ bitov, potrebuje $|CST| = O(\epsilon^{-1} n \log |\Sigma|)$ bitov. Pri tem je ϵ poljubna konstanta, ki ima vrednost $0 < \epsilon < 1$.*

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

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

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

Primer kompaktnega priponskega drevesa je prikazan na Sliki 15. Poleg kompaktne predstavitve (na desni strani slike), sta za lažjo berljivost in primerjavo prikazana tudi ekvivalentno priponsko drevo ter ekvivalentno priponsko polje (na levi strani slike). Prikazana podatkovna struktura potrebuje približno 12 B (natančno potrebuje 97 bitov). Pri tem so cela števila zapisana z $\log n$ biti. Ekvivalentno priponsko drevo potrebuje vsaj 18 B (oziroma 142 bita), priponsko polje pa potrebuje vsaj 9 B (oziroma 36 bitov, pri čemer priponsko polje, ki simulira priponsko drevo potrebuje približno 12 B oziroma 90 bitov). Pri tem nobena podatkovna struktura ne vključuje dodatnih podatkovnih struktur za opravljanje operacij v konstantnem času. Nekompaktne podatkovne strukture po navadi ne uporabljajo zapisa celih števil z $O(\log n)$ biti, kot je bilo uporabljeno v teh kalkulacijah, ampak se uporablja vsaj 1 B (8 bitov), po navadi 4 B, kar pa dodatno poveča razliko.

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 nadaljujemo z gradnjo in poizvedbami nad CST, 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 kompaktnega 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[1].

5.1 POIZVEDBE

Kompaktna predstavitev priponskega drevesa je ekvivalenta priponskemu drevesu, zato mora podpirati iste poizvedbe nad besedo T, ki jih podpira priponsko drevo. Najbolj preprosta poizvedba je $prisotnost(T, P)$. V kompaktnem priponskem drevesu se prisotnost vzorca P išče s pomočjo vzvratnega iskanja (angl. *Backward Search*) vzorca v priponskem polju SA. Vzvratno iskanje za predhodno predstavljeno kompaktno priponsko polje je prikazano na Algoritmu 6 in potrebuje $O(mt_\Psi)$ časa, da se izvrši. Za drugačno implementacijo kompaktnega priponskega polja, se morda mora nadomesti vrstico 6 v Algoritmu 6 z binarnim iskanjem nad Ψ_c in zato je potrebno $O(m \log nt_\Psi)$ časa, oziroma se uporablja drugi algoritem za iskanje vzorca. Algoritem vrne interval v priponskem polju, zato je potrebno 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 predstavljeno kompaktno priponsko polje, pa se poizvedba izvrši v času $O(m)$.

Algoritem vzvratnega iskanja, ki je predstavljen na Algoritmu 6, preveri prisotnost vzorca v priponskem polju, tako da se v koraku k najde interval pripon $[s, e]$ v priponskem polju, ki se začne z znakom $P[k]$ in se nadaljuje z nizom $P[k + 1 : m]$. Prvi interval $[s, e]$ predstavlja vse pripone, ki se začnejo z znakom $P[m]$ [6]. Na primer če želimo preveriti $prisotnost($»KOKOŠ$«, »KO«$)$, lahko to storimo s kompaktnim priponskim drevesom prikaznem na Sliki 15. Začetni interval $[s, e] = [4, 5]$, pri čemer je $C[P[m]] = C[O]$ oziroma je $C[3]$, saj je $S[3] = O$. V edinem koraku zanke je interval $[s', e']$ enak $[1, 2]$, saj je $c = K$ oziroma je $c = 2$, saj je $S[2] = K$. Postopek za Vrednost s' je izračunana kot $rang_1(B_2, 3) + 1$ in s pomočjo Slike 15 izračunamo, da je

[1]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.

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

```
(s, e) = (C[P[m]] + 1, C[P[m] + 1])
za i = (m − 1)..1
    če s > e potem
        return [−1, −1]
    c = P[i]
    (s′, e′) = (rang_1(B_c, s − 1) + 1, rang_1(B_c, e))
    (s, e) = (C[c] + s′, C[c] + e′)
če s > e potem
    return [−1, −1]
return [s, e]
```

$s' = 0 + 1 = 1$. Podobno lahko prevrimo tudi za $e' = rang_1(B_2, 5) = 2$. Zadnji korak izračuna je novi interval $[s, e] = [C[2] + 1, C[2] + 2] = [1 + 1, 1 + 2] = [2, 3]$. Ker s ni večji od e, vzvratno iskanje vrne $[2, 3]$, kar pomeni, da je vzorec prisoten v besedi.

Naslednja poizvedba je *številoPonovitev*(T, P), ki vrne število pojavov vzorca P v besedilu. V kompaktnem priponskem drevesu, pa je poizvedba ponovno implementirana s pomočjo vzvratnega iskanja, prikazanega na Algoritmu 6. Poizvedba *številoPonovitev*(T, P) je implementiran kot razlika $e - s + 1$, kjer s predstavlja prvo pripono, ki se začne z vzorcem P, in e je zadnja tako pripona, torej je razlika $e - s + 1$ število pripon. Torej operacija ponovno potrebuje $O(mt_\Psi)$ oziroma $O(mt_\Psi \log n)$ časa, da se izvrši.

Zadnja predstavljena poizvedba pa je *seznamPojavov*(T, P), ki vrne vse indekse pojavov vzorca P v besedi T. V kompaktnem priponskem drevesu pa je poizvedba ponovno implementirana s pomočjo vzvratnega iskanja, prikazanega na Algoritmu 6. Z vzvratnim iskanjem se naračuna interval v priponskem polju $SA[s, e]$. Za pridobitev položajev ponovitev vzorca v besedilu, je potrebno ustvariti seznam $[SA[s], SA[s + 1], \ldots, SA[e]]$, kar zahteva še dodatnih $occ = e - s$ korakov, pri čemer vsak korak potrebuje $O(t_{SA})$ časa, da se izvrši. Torej poizvedba *seznamPojavov*(T, P) potrebuje $O(mt_\Psi + occ \cdot t_{SA})$ oziroma $O(mt_\Psi \log n + occ \cdot t_{SA})$ časa. S predhodno predstavljeno implementacijo kompaktnega priponskega drevesa pa je potrebno $O(m + occ \cdot \log n)$ 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 τ in LCP polje odvečni podatkovni strukturi. To je res zgolj za osnovne poizvedbe nad besedilom T, ki so lahko izvršene zgolj z uporabo kompaktnega priponskim polje v enakem času. Poizvedbe, kot so najdaljši ponavljajoči podniz, najdaljši palindrom, ki je implementirana s pomočjo priponskega drevesa konkatenacije obrata T'_z vhodne besede T_z, $T = T_z\#T'_z\$$, in najdaljši skupni niz besed T_1 in T_2, ki je implementirana s pomočjo priponskega drevesa konkatenacje obeh besed $T = T_1\#T_2\$$. Na primer poizvedbe najdaljši ponavljajoči podniz je podniz $T[SA[i], SA[i] + sd(v)]$, pri čemer i je skrajno levi list poddrevesa s korenom v notranjem vozlišču v in velja, da je $LCP[i]$ največji element v LCP polju, torej zahteva $O(nt_{SA})$ časa. Operacije $sd(v)$ ni potrebno 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 $T[SA[j], SA[j] + sd(u)]$, pri čemer je j skrajno levi list poddrevesa s korenom v notranjem vozlišču $u = sl(v)$. Za izračunat le tega pa je potrebo $O(nt_{SA} + t_\Psi)$ časa ter se uporabijo vse tri podatkovne strukture kompaktnega priponskega drevesa [3, 5, 6].

5.2 GRADNJA

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

Kompaktno priponsko drevo je mogoče izgradi neposredno iz vhodnega besede T. Pri tem so potrebne dodatne podatkovne strukture za izgradnjo posamičnih komponent. Tudi te dodatne podatkovne strukture so kompaktne podatkovne strukture, zato bo zgrajeno v kompaktnem prostoru. Izgradnja kompaktnega prionskega drevesa poteka v sledečih treh korakih:

1. izgradnja kompaktnega priponskega polja SA iz besede T,

2. izgradnja kompaktne predstavitve polja LCP iz kompaktnega priponskega polja SA in besede T,

3. izgradnja kompaktne predstavitve topologije drevesa τ iz LCP polja.

Izgradnja CSA. Prva zgrajena podatkovna struktura je kompaktno priponsko polje SA. Obstajajo različne implementacije kompaktnega priponskega polja, zato ni mogoče predstaviti splošnega algoritma za izgradnjo le tega. V nadaljevanju bo predstavljen

algoritem izgradnje predhodno predstavljene implementacije CSA. Ker je priponsko polje implementirano s pomočjo funkcije Ψ (shranjena v istoimenskem polju), je potrebno naračunati polje Ψ. Funkcijo Ψ se gradi od leksikografsko najmanjše pripone do največje pripone s pomočjo polja $L[1, n]$, pri čemer je $L[i] = T[SA[i] - 1]$ ter $T[0] = T[n] = \$$. Polje L se lahko zgradi v manjših delih, zato se priponsko polje izgradi v delih dolžine b. Za $b = n/\log n$ je potrebnih $o(n)$ bitov dodatnega prostora za shraniti del priponskega polja in $O(n \log n)$ časa za zgraditi kompaktno priponsko polje [6].

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 na SA_k za $1 \leq k \leq b$. Sledi izračun pripon, ki sodijo v določen del priponskega polja SA_k. Pripone v tem delu se leksikografsko uredijo, najpogosteje z uporabo korenskega urejanja. V k-tem koraku izgradnje je polje L izgrajeno za indekse od 1 do $kb - 1$. Vrednost celice $kb + j$ se izračuna kot $L[kb + j] = T[SA[j] - 1]$ [6].

Ko je polje L izračunano, se kompaktno predstavitev polja Ψ 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 ni potrebno zapisati črke $c = T[A[j] - 1]$ v polju L, ampak se lahko neposredno zapiše enico v celico $B_c[kb + j] = 1$ [6].

Vzporedno z izdelavo kompaktne predstavitve poja Ψ se lahko naračuna tudi polja vzorcev SA_S in SA_S^{-1}, saj se za izgradnjo sprotno naračunava priponsko polje SA. Torej za izgradnjo poji vzorcev ni potrebno dodatnega časa. 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.

Izgradnja LCP strukture. 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 se lahko sprotno vpiše enico na mesto izračunane vrednosti $PLCP[i] + 2i$ v bitnem polju H [6].

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

predstavljeno implementacijo $t_{SA} = O(\log n)$, in potemtakem je za izgradnjo potrebno $O(nt_{SA})$ ali za predhodno predstavljeno implementacijo $O(n \log n)$.

Izgradnja topologije drevesa. Zadnja izgrajena podatkovna struktura je topologija priponskega drevesa τ. 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 τ je podobna algoritmu Kasai idr. [14], ki je bil predhodno predstavljen v podpoglavju 4.3.

To je storjeno s sprehodom 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 črkovni globini $sd(v) = LCP[i]$. Če je tako vozlišče v že v drevesu potem list i postane njegov desni otrok. Sicer $sd(v) < LCP[i]$ in obstaja vozlišče u, ki je lahko $i-1$-vi list, za katerega velja $sd(u) > LCP[i]$. V tem primeru se ustvari novo vozlišče v', ki postane desni otrok od v ter ima dva otroka, in sicer levega otroka u ter desnega otroka listi i. Na ta način se izgradi priponsko drevo zgolj iz priponskega polja SA in LCP polja. Predstavljena metoda je implementirana z uporabo sklada, ki hrani vozlišče v ter črkovno globino $sd(v)$. Na skladu se ne shrani kazalec na vozlišče, ampak se shrani predstavitev poddrevesa s korenom v vozlišču v z zaporedjem uravnoteženih 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 črkovno dolžino $sd(koren()) = 0$. Na sklad je dodanih največ $n-1$ vozlišč, ker ima priponsko drevo največ $n-1$ notranjih vozlišč. Da bo na koncu izgradnje sklad prazen in topologija drevesa τ pravilna, se sklad izprazni in na ta način se pridobi pravilno topologijo τ, saj vsa vozlišča na skladu ne shranjujejo topologije za njihovo skrajno desno poddrevo, ker le to še raste. Torej vsakič, ko se vozlišče sname iz sklada, se topologija poddrevesa doda staršu vozlišča, ki je vozlišče na vrhu sklada [6].

Izgradnja topologije drevesa poteka v času $O(nt_{SA})$ oziroma $O(n \log n)$ za predhodno predstavljeno implementacijo CSA. V vsakem koraku je v topologijo drevesa dodan nov list. Za vsak list je lahko dodano novo vozlišče na sklad, če za črkovno dolžino vozliča 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 τ.

Izrek 5.9. *Časovna zahtevnost izgradnje kompaktnega priponskega drevesa za vhodno besedo 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 za predstavljeno implementacijo CSA, torej je za n pripon potrebno $O(n \log n)$ časa.

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

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

6 EMPIRIČNA EVALVACIJA OPERACIJI NAD PRIPOSKIMI DREVESI

Do sedaj so bile teoretično predstavljene različne implementacije prionskih dreves. V tem poglavju pa bodo te implementacije priponskih dreves empirično primerjane med seboj. In sicer se bodo testirale sledeče podatkovne strukture:

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

Za vsako od naštetih podatkovnih struktur bodo testirane tri strari: velikost podatkovne strukture, čas potreben za gradnjo in čas potreben za poizvedbe z podatkovno strukturo.

To poglavje bo razdeljeno na tri dele, is sicer prvo bo predstavljeno testno okolje, metoda testiranja ter vhodna besedila. Za tem bodo predstavljeni rezultati testov. V zadnjem delu poglavja pa bo narejena razprava rezultatov primerjav.

6.1 EKASIPERIMENTALNO OKOLJE

Pred se lahko izdela primerjavo podatkovnih struktur je potrebno predstaviti okolje, v katerem bo izdelana primerjava, metoda uporabljena za izdelavo primerov ter testni primeri, ki so uporableni za izdelavo primerjve.

6.1.1 Stroj, operaciski sistem in metoda testiranja

Za izdelavo primerjva podatkovnih struktur bo uporabljen računalnik s procecorjem Intel Core i3 5005U z dvema jedroma in štirimi nitmi ter s taktom $1,9$ GHz. Računalnikima na razpolago 4 GB delovnega pomnilnika, od katerega je ob zagonu zasedengea $1,35$ GB z operaciskim sistemom ter emulatorejm terminala (angl. *terminal emulator*). Poleg delovnega pomnilnika ima računalnik še 8 GB Swap razdelka na trdem disku, ki

je namenjen shranjevanju neuporabljenih pomnilniških strani v primeru prezasedenega delovnega pomnilnika in posledično sproščanju le tega. Operaciski sistem računalnika je Fedora 42 in uporabljen Linux kernel je 6.14.11-300.

Za vsako primerjano podatkovno strukturo bodo izmerjene tri strari: prostor, ki ga zasede na pomnilniku, čas potreben za izgradnjo podatkovne strukture ter čas potreben za izvršitev poizvedb v podatkoni strukturi. Struktura primerjava je prikazana s psevdokodo na Algoritmu 7. Vsak test v primerjavi se izvede 5-krat, da se zmanjša vpliv drugih procesov na rezultate tetiranja. Iz psevdo kode opazimo, da primerjva poizved bo storjena za poizvedbo $prisotnost(T, P)$, saj je pogosto vprašanje v biologiji prisotnost gena (vzorcev) v DNK sekvenci (vhodna beseda), ne pa natančen položaj tega gena ali števila ponovitev gena v DNK sekvenci. Poizvedba je bila izdelana za vzorce velikosti 5, 50, 500 in $\log |T|$ (na psevdokodi $\log i$). Rezultati testiranja so shranjeni v CSV datoteko in vsaka vrstica datoteke, shrani sledeče podatke: podatkovna struktura, velikost vhodne besede, čas izgradnje, čas iskanja vzorca dolžine 5, čas iskanja vzorca dolžine 50, čas iskanja vzorca dolžine 500, čas iskanja vzorca dolžine $\log |T|$ ter velikost podatkovne strukture. Velikost bo dodana ročno iz podatkov priboljenih s pomnilniškim profilerjem (angl. *memory profiler*).

Primerjava podatkovnih struktur je izdelana v programskem jeziku C++[1]. Zato se čase potrebne za izgradnjo struktur ter poizvedb v njih izmeri z beleženjem trenutnega časa pred začetkom, ki ga shranimo v spremenljivko `start`, in po koncu operacije, ki pa ga shranimo v spremenljivko `stop`. Meriteve so implementirane s funkcijo `high_resolution_clock::now()`, ki je del standardne knjižnice programskega jezika C++ in vrne natančen čas trenutka, v katerem jefunkcija izvedena. Čas izvajanja se lahko naračuna kot razlika med časom ob koncu izvajanja in časom ob začetku izvajanja operacije, kar se v C++ stori s funkcijo `duration_cast<nanoseconds>(stop - start).count()`. Za izgradnjo podatkovnih struktur bodo uporabljeni sledeče knjižnice. Priponsko drevo uporabila implementacijo iz knjižnice [12], ki implemenira Ukkonenov algoritem [1]. Kompaktna priponska drevesa so bila izdelana s implementacijo iz knjižnice SDSL [11]. Priponska polja pa so bila implementira s pomočjo knjižnice [30], ki izgradi priposnko polje v $O(n)$ časa z algorimom, ki sta ga predstavila Timoshevskaya in Feng [31]. Algoritem je izboljšava predhodno predstavljenga aligoritma od Ko in Aluru [25]. Podatkovna struktura LCP polje, ki je bila predstavljena v podpoglavju 4.1 in sicer podatkovna struktura $Q - LCP$, izračunana v času $O(n)$. Knjižnica [30] sicer podpira izgradnjo LCP strukture, ki je bila predstavljena v podpoglavju 4.3, ampak nima dodatne podatkovne strukture Fischer-Heun za operacijo *rmq* v konstantnem času [28].

Poleg časovne zahtevnosti izgradnje in pozved v podatkovnih strukturah nas za-

[1]Koda je dostopna na povezavi `https://github.com/GioGiou/MagisterskaNalogaKoda`.

Algoritem 7: Psevdokoda primerjave indeksov

Vhod: Vhodno beseda T, število znakov v najdaljšem testiranem nizu i_{max}

Izhod: CSV datoteka z izračunanimi časi

```
1  i ← 500
2  dokler i ≤ i_max
3      za S ∈ {ST, SA, SA + LCP, CST}
4          za k = 1 ... 5
5              start ← čas()
6              Izgradi(S, T[1 : i − 1] · $)
7              konec ← čas()
8              t_Izg ← konec − start
9              dokler j ≤ 500
10                 start ← čas()
11                 prisotnost(S, T[i : i + j])
12                 konec ← čas()
13                 j ← j ∗ 10
14                 t_j ← konec − start
15             j ← log i
16             start ← čas()
17             prisotnost(S, T[i : i + j])
18             konec ← čas()
19             t_log ← konec − start
20             Shrani(S, i, t_Izg, t_5, t_50, t_500, t_log)
21             Izbriši(S)
22     i ← i + 500
```

nima, koliko prostora posaminčna podatkovna struktura zasede na delovnem pomnilniku. In sicer nas zanima največji zaseden prostor v času izgradnje posamične podatkovne strukture. Le ta bo izmerjen z uporabo pomnilniškega profilerja. Uporabljen bo pomnilniški profiler Bytehound [20], saj omogoča grafični prikaz porabe pomnilnika v času izvajanja programa.

Ker je primerjava izdelana v C++, je potrebno kodo prevesti v izvršljivo datoteko. To je strojeno s prevajalinkom GCC 15.1.1. Program je bil preveden s sledečim ukazom:

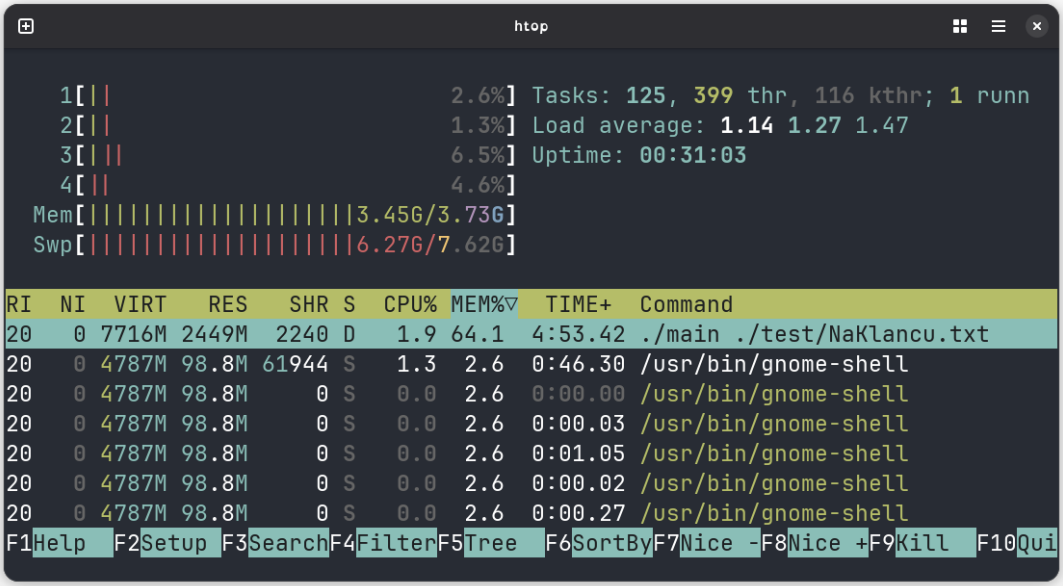
```
g++ -std=c++11 -O3 -DNDEBUG -I ./include -L ./lib //
main.cpp -o main -lsdsl -ldivsufsort //
-ldivsufsort64 -lsuffix -lsais
```

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`, `-lsais` 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 kompaktnega priponskega drevesa [3]. Ker se uporablja profiler, je potrebo pred začetkom testiranja indeksov besed 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 se jo shrani v sistemsko spremenljivko `MEMORY_PROFILER_LOG`. Naslednja vrstica pa požene program ter se poda kot prvi parameter ime datoteke s testnimi podatki. 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 je deljeni objekt profiler, kar mu omogoča dostop do programa in lažje beleženje zasedenega delovnega spomina.

Dolžina najdaljše vhodne besed i_{max} je bila sprva nastavljena na 4000000 znakov. Med testiranjem priponskega drevesa besede dložine 2048000 znakov se je opazilo, da čas potreben za izgradnjo prvega priponskega drevesa presega 5 minut. Po petih minutah od začetka izgradnje drevesa lahko poazimo, da je delovni pomnilnik skoraj v celoti zaseden ter je Swap razdelek zaseden že z 6 GB pomnilniški strani. Pri tem je računalnik neodziven in proces je v neprekinjenem spanju (angl. *Uninterruptible sleep* ali stanje D). Na Sliki 16 je prikazan upravljalnik opravil Htop v času izgradnje priponskega drevesa za besedo dolžine 2048000 znakov. Proces v modri vrstici predstavlja program za testiranje čas izgradnje in pozvedb ter postorske zahtevnosti indeksov besed. 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 priponskega drevesa za besedo dolžine 2048000 znakov sem se odločil, da se proces ubije. Posledično se je znižala velikost besede za izgradnjo zadnjega indeksa na 1024000 znakov. Med testiranjem sem opazil tudi, da se LCP polje ne izgradi za besede dolžine 1024000 znakov, zato sem se odločil, da se ta test v tem koraku izpusi.



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

6.1.2 Testni primeri

Predstavljeno tesno metodo bomo pognali nad dvemi vhodnimi besedili prikazanih v Tabeli 4. Prva testirana beseda je Cankarjev kratki roman Na klancu [10]. Roman je bil izbran, saj predstavlja tipično vhodno besedo naravnega jezik, ki je v tem primeru slovenščina. Abeceda takih besed sestoji iz črk naravenga jezika (velikih in malih), presledkov in ločil. Pri tem se je pojavil probelm, saj slovenščina ne uporablja ASCII znake. Zato je potrebno pred samim začetkom testiranja vhodno besedo 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.

Druga testirana beseda pa je daljša DNK sekvenca [9], ki je zlepek različnih DNK sekvenc. Izbrana je bilam kot primer vhodne besede uporabljene v bioinformatiki. Ta vhodna beseda je bila predhodno uporabljena za testiranje implementacije kompaktnega priponskega drevesa od Välimäki idr. [3]

Tabela 4: Primerjava besedil, ki bodo uporabljena za primerjavo različnih implementacij priponskih dreves

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

Podalševanje besedila. V drugem stolpcu Tabele 4 so prikazane dolžine besed, ki bodo uporabljen za testiranje. Pri tem se opazi, da nekatera besedila so krajša kot 2048500 znako, ki so potreni za izgradnjo indeksov in za izdelavo vzorcev testiranja. Zato je treba te besede podalšati. 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 podnize besedila ter naredi stik med začetno besedo in podnizom. 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 na Algoritmu 8. Pri tem metoda predpostavi, da je besedilo dolgo vsaj 3000 znakov. Ta metoda je primerna za podaljševanje daljših besedil, sicer pa je možno metodi spremeniti parameter i in tako prilagoditi metodo drugim besedilom.

Algoritem 8: Metoda podaljševanja vhodnega besedila

Vhod: Vhodno besedilo T, želena velikost s_{max}
Izhod: Besedilo T_0

```
1 T_0 ← T
2 i ← 500
3 while |T_0| < s_max do
4 |   T_0 ← T_0 · T[i : 6i]
5 |   i ← i + 500
6 |   while 6i > |T| do
7 |   |   i = i/4
8 vrni T_0[1 : s_max]
```

Predlagana metoda podaljša besedo na velikost s_{max}. Ta velikost je lahko dolžina najdaljšega besede ali pa je poljubna vrednost, bodisi manjša od največjega besede bodisi večja. Če je beseda daljša od velikosti $s_{max} < |T|$, bo predlagana metoda skrajšala besedo na $|T_0| = s_{max}$. Za potrebe testiranja sem se odločil, da je $s_{max} = 25000000$, kar je več kot zadostna dolžina.

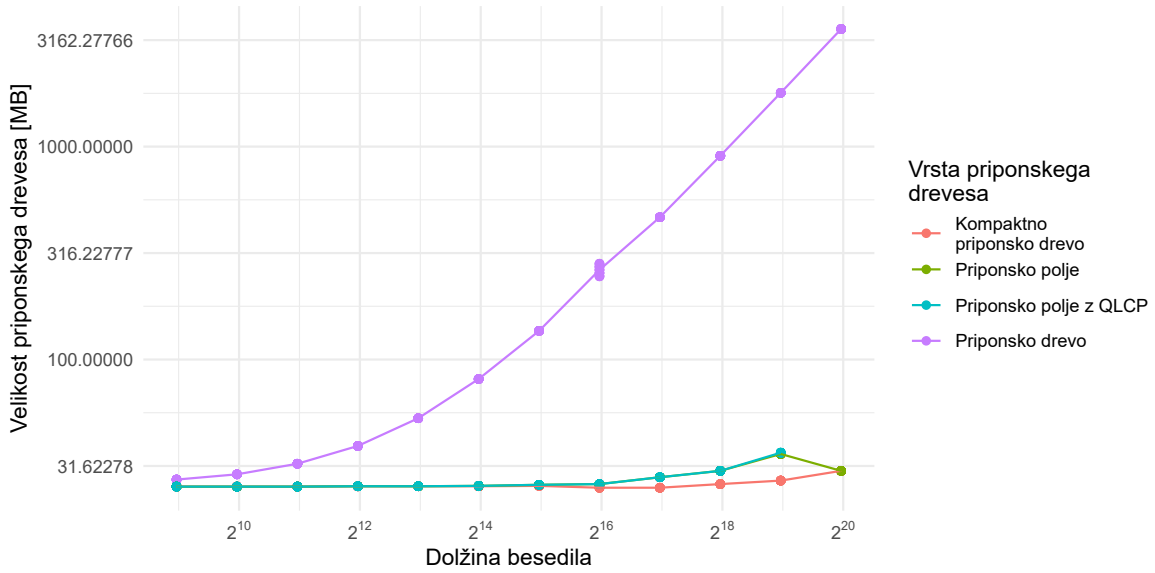
6.2 PRIMERJAVA

V tem podpoglavju bodo predstavljeni rezultati testiranja različnih indekso besede, ki je bil izdelan z metodo in računalnikom predstavljenim v podpoglavju 6.1. Metoda meri prostor, ki ga zasede indeks, čas potreben za izgradnjo indeksov in čas potreben za različne poizvedbe z indeksom. Meritve lahko ločimo na dva dela, in sicer: meritve vezane na gradnjo, ki sta čas izgradnje in velikost indeksa besede, ter poizvedbe z indeksom. Zato bo sledeče podpoglavje tudi razdeljeno na dva dela. V vsakem delu bodo ločeno predstavljeni rezultati testiranja za vsako testno besedo iz Tabele 4.

6.2.1 Gradnja

Testiranje se indekso se začne z izgradnjo indeksov. Pri tem smo merli čas potreben za izgradnjoj indeksov za besede različnih dolžin. Enkrat ko je indeks izgrajen, se lahko izmeri tudi njegovo velikost na pomnilniku.

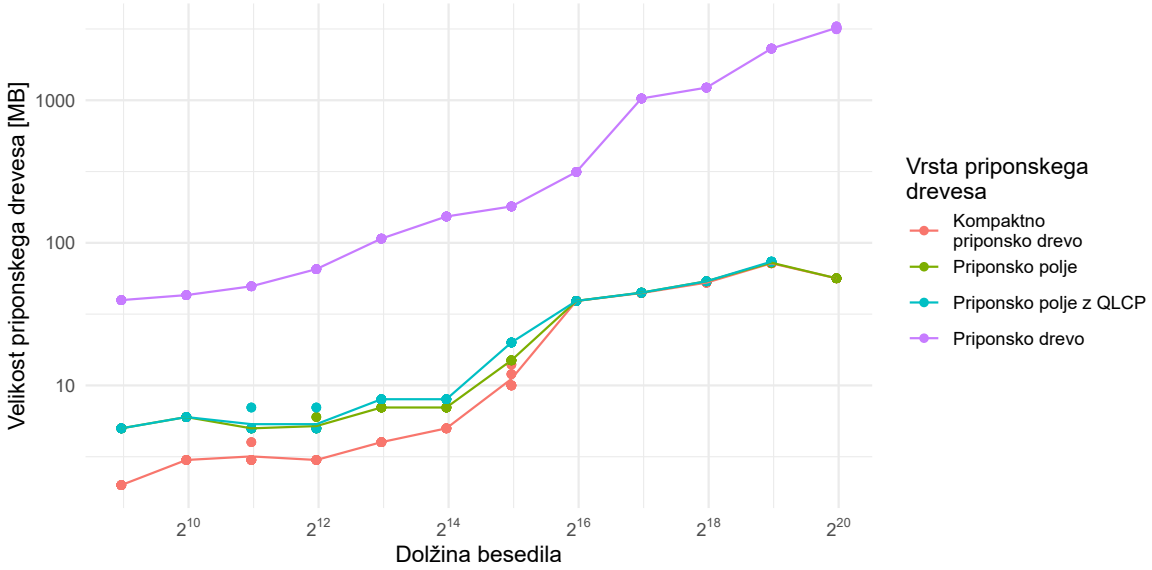
Prostorska zahtevnost podatkovne strukture. In če se začne lih z velikostjo indeksov na pomnilniku, sto te meritve prikazane na Sliki 17 za DNK sekvenco in na Sliki 18 za roman Na klancu. Iz obeh testiranj se vidi, da je prostorska zahtevnost priponskega drevesa večja, kot prostorska zahtevnost ostalih indeksov. Pri tem vidimo tudi, da prostorska zahtevnost linearno raste z velikostjo besede, za vse 4 testne primere.



Slika 17: Graf velikosti indeksov izgrajenih iz besed različni velikosti. Vhodna beseda je DNK sekvenca.

Iz rezultatov testiranja za DNK sekvenco, Slika 17, vidimo, da vse podatkovne strukture linearno rastejo. Pritem pa priponsko polje, priponsko polje z LCP poljem

in kompaktno priponsko drevo potrebujejo bistveno manj prostora.



Slika 18: Graf velikosti indeksov izgrajenih iz besed različni velikosti. Vhodna beseda je roman Na klancu.

Podobno se lahko vidi tudi za roman Na klancu, Slika 18. Pritem pa se vidi, da je razlika v zasedem prostoru nižja in konstantna skozli celotno izvajanje testa.

Časovna zahtevnost izgradnje.

6.2.2 Poizvedbe

Časovna zahtevnost poizvedbe za vzorec dolžine 5.

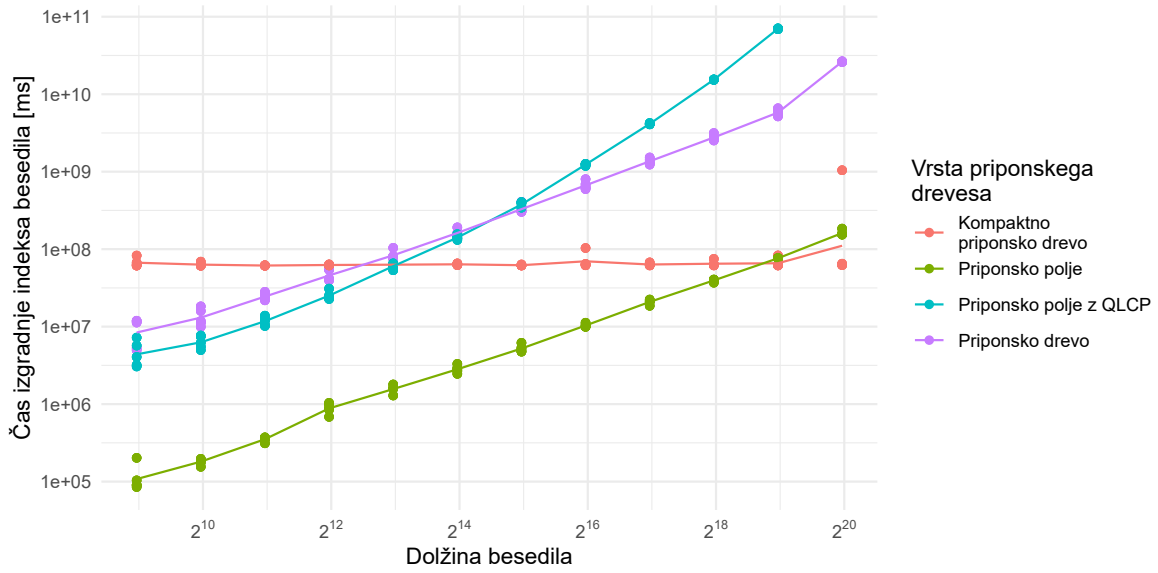
Časovna zahtevnost poizvedbe za vzorec dolžine 50.

Časovna zahtevnost poizvedbe za vzorec dolžine 500.

Časovna zahtevnost poizvedbe za vzorec dolžine $\log n$**.**

6.3 RAZPRAVA

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



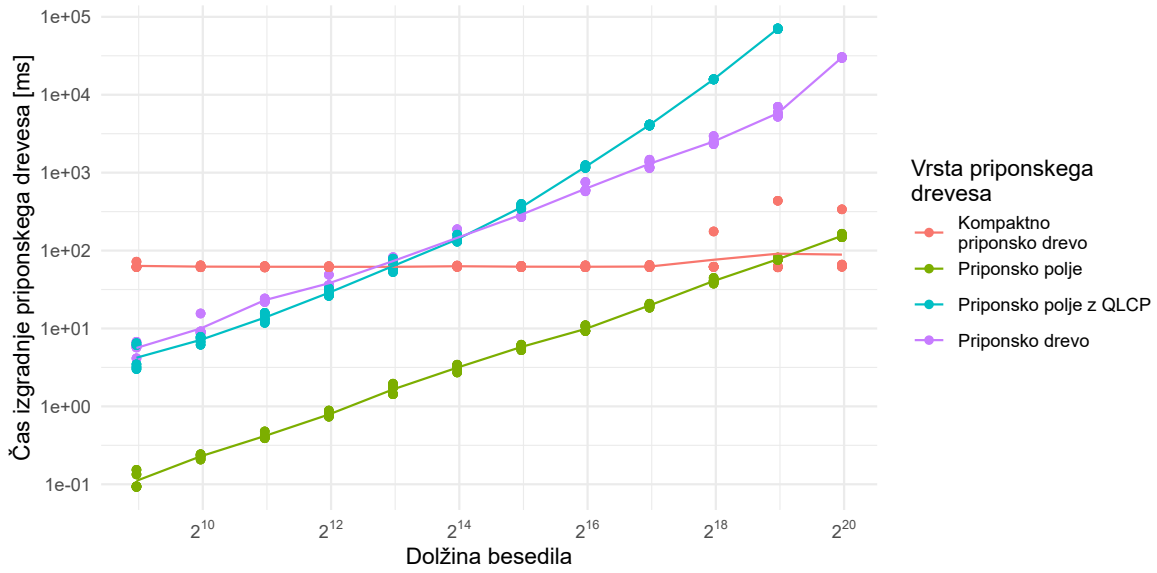
Slika 19: Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je DNK sekvenca.

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.

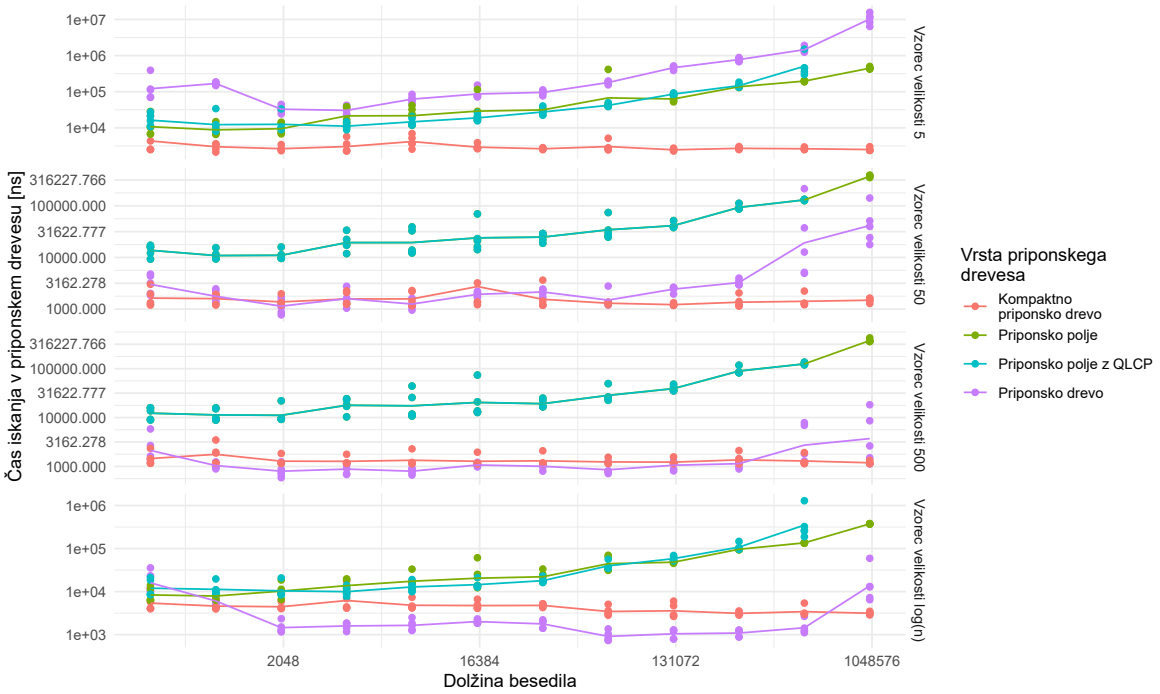
Pri primerjavi rezultatov obeh testiranj, testiranje nad DNK sekvenco in testiranje nad besedilom iz naravnega jezika, se lahko opazi, da so si rezultati zelo podobni. Edina bistvena razlika med obema testiranjema je skok v potrebnem času pri iskanju daljših vzorcev ter vzorcev dolžine $O(\log n)$ v 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

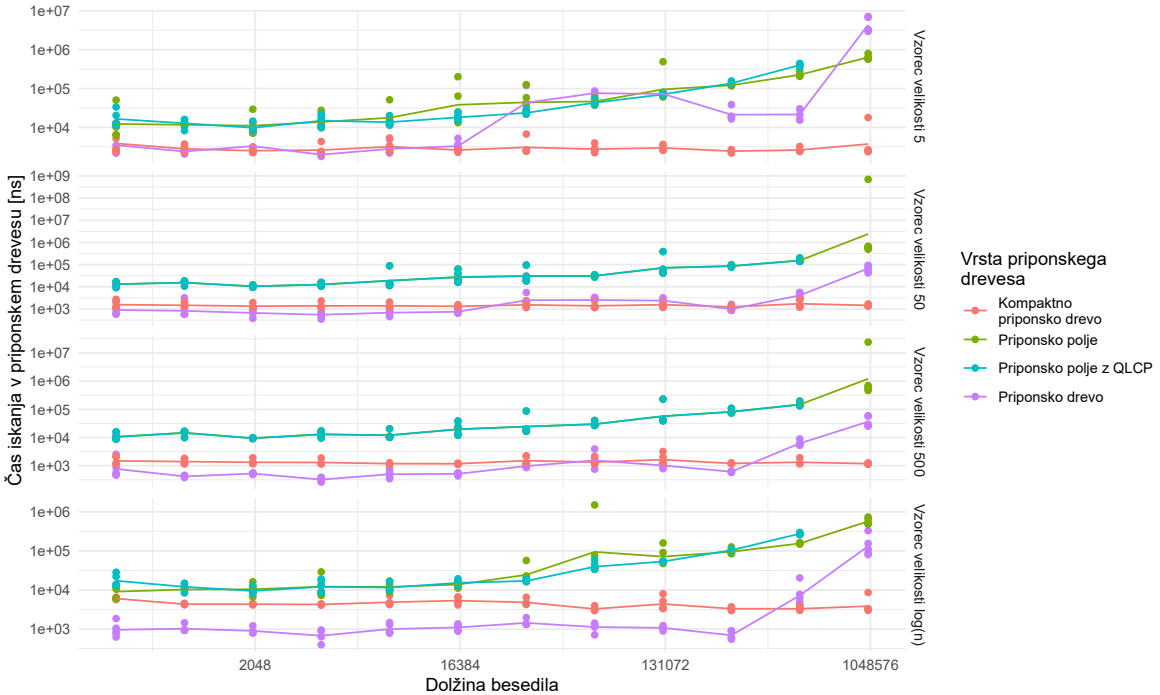


Slika 20: Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je roman Na klancu.

od Jacobsa [21] in Välimäki idr. [3] je lahko pojasnjena velika časovna zahtevnost pri izgradnji priponskega drevesa velikost 2048000 znakov.



Slika 21: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je DNK sekvenca.



Slika 22: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je roman Na klancu.

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 podatkovnikih 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 dvige 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 kompaktnega prionskega polja, ki je uporabljeno v kompaktnem priponskem drevesu. Če je kompaktno priponsko drevo implementirano s pomočjo časovno najbolj učinkovite implementacije kompaktnega 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 kompaktnega 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 kompaktnega priponskega drevesa v celoti storjen s kompaktnimi podatkovnimi strukturami.

Z empirično primerjavo so bile potrjene razlike v prostorski zahtevnosti ter v razliki časovnih 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 osebi 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 kompaktnega priponskega polja, ki zniža obe časovni zahtevnosti t_{SA} in t_{Ψ} 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 23, 28, 30, 32 in 60.)*

[2] K. Sadakane, Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems* 41 (2007) 589–607. *(Citirano na straneh 45, 46, 47, 48, 50 in 51.)*

[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 47, 55, 62, 63, 67 in 68.)*

[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 23, 28 in 32.)*

[5] P. Weiner, Linear pattern matching algorithms. V *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, 1973, 1–11. *(Citirano na straneh 1 in 55.)*

[6] G. Navarro, *Compact Data Structures: A Practical Approach*, Cambridge University Press, 2016. *(Citirano na straneh 1, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 16, 18, 48, 49, 50, 53, 54, 55, 56 in 57.)*

[7] D. E. Knuth, J. H. Morris in V. R. Pratt, Fast Pattern Matching in Strings. *SIAM Journal on Computing* 6 (1977) 323-350. *(Citirano na strani 19.)*

[8] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997. *(Citirano na strani 19.)*

[9] *Pizza&Chili Corpus – Compressed Indexes and their Testbeds*, `https://pizzachili.dcc.uchile.cl/`. (Datum ogleda: 8. 10. 2024.) *(Citirano na straneh 63 in 64.)*

[10] I. Cankar, *Na Klancu*, Genija, 2012. *(Citirano na straneh 63 in 64.)*

[11] S. Gog, T. Beller, A. Moffat in M. Petri, From Theory to Practice: Plug and Play with Succinct Data Structures. V *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, 326–337. *(Citirano na strani 60.)*

[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 60.)*

[13] E. Miller, *Genome — Knowledge Hub*, `https://www.genomicseducation.hee.nhs.uk/genotes/knowledge-hub/genome`. (Datum ogleda: 30. 10. 2024.) *(Citirano na strani 45.)*

[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 38, 40 in 57.)*

[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 47.)*

[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 48 in 51.)*

[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 48 in 51.)*

[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 52.)*

[19] D. Harel in R. E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing* 13 (1984) 338-355. *(Ni citirano.)*

[20] Koute, *koute/bytehound: A memory profiler for Linux.*, `https://github.com/koute/bytehound`. (Datum ogleda: 30. 10. 2024.) *(Citirano na strani 61.)*

[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 straneh 67 in 68.)*

[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 36 in 39.)*

[25] P. Ko in S. Aluru, Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms* 3 (2005) 143-156. *(Citirano na straneh 39 in 60.)*

[26] M. I. Abouelhoda, S. Kurtz in E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* 2 (2004) 53-86. *(Citirano na straneh 40, 41 in 43.)*

[27] D. Knuth, *Art of Computer Programming, The: Combinatorial Algorithms, Volume 4A*, Addison-Wesley Professional, 2011. *(Citirano na strani 6.)*

[28] J. Fischer in V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array. V *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, 2007, 459–470. *(Citirano na straneh 40 in 60.)*

[29] A. Brodnik, *Searching in Constant Time and Minimum Space*, Ph.D. dissertation, University of Waterloo, 1995. *(Citirano na strani 6.)*

[30] I. Grebnov, *GitHub - IlyaGrebnov/libsais: The libsais library provides fast linear-time construction of suffix array (SA), generalized suffix array (GSA), longest common prefix (LCP) array, permuted LCP (PLCP) array, Burrows-Wheeler transform (BWT) and inverse BWT based on the induced sorting algorithm with optional OpenMP support for multi-core parallel construction.*, `https://github.com/IlyaGrebnov/libsais`. (Datum ogleda: 30. 6. 2025.) *(Citirano na strani 60.)*

[31] N. Timoshevskaya in W. Feng, SAIS-OPT: On the characterization and optimization of the SA-IS algorithm for suffix array construction. V *IEEE 4th International Conference on Computational Advances in Bio and Medical Sciences, ICCABS*, 2014, 1-6. *(Citirano na strani 60.)*