UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Magistrsko delo

Kompaktna priponska drevesa

(Compressed Suffix Tree)

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

Koper, april 2025

Ključna dokumentacijska informacija

Ime in PRIIMEK: Jani SUBAN
Naslov magistrskega dela: Kompaktna priponska drevesa

Kraj: Koper
Leto: 2025

Število listov: 73 Število slik: 17 Število tabel: 6
Število referenc: 21
Mentor: prof. dr. Andrej Brodnik
Somentor: izr. prof. dr. Rok Požar

UDK:
Ključne besede:
POPRAVI: Math. Subj. Class. (2010):
Izvleček:
POPRAVI:: Izvleček predstavlja kratek, a jedrnat prikaz vsebine naloge. V največ 250 besedah nakažemo problem, metode, rezultate, ključne ugotovitve in njihov pomen.

Key document information

Name and SURNAME: Jani SUBAN
Title of the thesis: Compressed Suffix Tree

Place: Koper
Year: 2025

Number of pages: 73 Number of figures: 17 Number of tables: 6
Number of references: 21
Mentor: prof. dr. Andrej Brodnik
Co-Mentor: izr. prof. dr. Rok Požar

UDC:
Keywords:
Math. Subj. Class. (2010):
Abstract:
Izvleček predstavlja kratek, a jedrnat prikaz vsebine naloge. V največ 250 besedah nakažemo problem, metode, rezultate, ključne ugotovitve in njihov pomen.

Kazalo vsebine

Kazalo preglednic

Kazalo slik in grafikonov

Seznam kratic

angl.	Angleščina
slo.	Slovenščina
idr.	in drugi
DNK	Deoksiribonukleinska kislina
KMP	Knuth–Morris–Pratt
ST	angl. *Suffix Tree* (slo. Priponsko drevo)
CST	angl. *Compressed Suffix Tree* (slo. Kompaktno priponsko drevo)
LOUDS	angl. *Level Order Unary Degree Sequence* (slo. Zaporedje eniških zapisov stopenj vozlišč po plasteh)
BP	angl. *Balanced Parentheses* (slo. Uravnoteženi oklepaji)
rmM	angl. *range minimum-maxsimum)*
rmq	angl. *range minimum querry)*
rMq	angl. *range maxsimum querry)*
DFUDS	angl. *Depth Frst Unary Degree Sequence* (slo. Zaporedje eniških zapisov stopenj vozlišč v globino)
SA	angl. *Suffix Array* (slo. Priponsko Polje)
CSA	angl. *Compressed Suffix Array* (slo. Kompaktno priponsko polje)
LCP	angl. *Longest Common Preffix* (slo. Najdaljša skupna predpona)
BWT	angl. *Burrows–Wheeler Transform* (slo. Burrows–Wheelerjeva preslikava)
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 besedah je iskanje vzorcev v njih. V procesiranju naravnih jezikov se ta problem pretvori na iskanje vseh ponovitev besede v besedilu. Medtem ko se v bioinformatiki ta problem pretvori na preverjanje prisotnost specifičnih genov ali drugih DNK sekvenc v genomu.

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

Namen magistrske naloge je preučevanje vpliva kompaktne predstavitve na izgradnjo priponskih dreves in na operacije nad njimi. V nalogi bodo predstavljene časovne zahtevnosti osnovnih operacij kot tudi empirična primerjava različnih implementacij priponskih dreves.

1.1 STRUKTURA

Magistrska naloga je razdeljena na tri dele. V drugem 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 $O(n)$.

V tretjem 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 prikazane različne kompaktne prestavitve dreves in algoritem za izgradnjo kompaktnega pripon-

skega drevesa, ki skozi celotno izgradnjo ohranja kompaktnost podatkovne strukture.

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

2 PRIPONSKA DREVESA

Priponska drevesa so posebna implementacija številskega drevesa, pri čemer vsak list ne predstavlja posamične besede, ampak 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 na pomnilniku kot polje črk `T[1..n]`, ampak zakodira tudi njegovo strukturo. Zato se pogosto uporabi za indeksiranje besede in posledično iskanja vzorcev v njej. Vzorec $P[1, m]$ se nahaja v besedi T, če obstaja podniz $T[i, j]$, za katerega velja $P[1, m] = T[i, j]$. Z uporabo indeksa je iskanje prisotnosti vzorca P dolžine m v besedi primerljivo s KMP algoritmom, ki potrebuje $O(n+m)$ časa za preveriti, ali se vzorec nahaja v besedi. Za razliko od KMP algoritma je prednost uporabe priponskega drevesa, da lahko iščemo različne vzorcev v isti besedi T. Za najti več vzorcev v priponskem drevesu je potrebno: $O(n)$ časa za izgradnjo priponskega drevesa ter $O(m)$ časa za vsak iskani vzorec. Pri iskanju več vzorcev v besedi T algoritem KMP potrebuje $O(n + m)$ časa za vsak iskan vzorec [7, 8].

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

Definicija 2.1. Priponsko drevo nad nizom T dolžine n je številsko drevo, ki zadošča sledečim zahtevam:

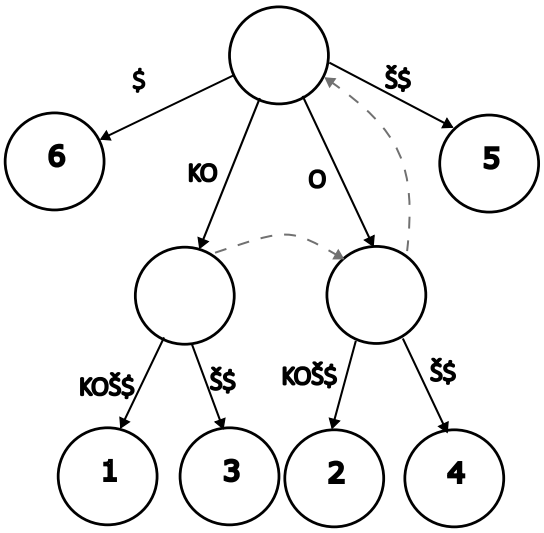
1. drevo ima natanko n listov oštevilčenih s števili med 1 in n,
2. vsako notranje vozlišče, razen korena, ima vsaj dva otroka,
3. vsaka povezava predstavlja neprazni podniz besedila T,
4. ne obstajata povezavi, ki se začneta v istem vozlišču in z istim znakom,
5. podniz, ki je pridobljen s stikom podnizov na poti od korena do lista i, predstavlja pripono $T[i, n]$ za vsak i, kjer je $1 \leq i \leq n$.

Definicija 2.2. Niz $\alpha \in \Sigma^*$ je podniz besedila T, ko obstaja tak $1 \leq i \leq n$, za katerega velja, da je $\alpha = T[i, i + |\alpha|]$.

Definicija 2.3. Stik nizov α in β je niz $\gamma = \alpha \cdot \beta$, pri čemer je $\alpha = \gamma[1, |\alpha|]$ in $\beta = \gamma[|\alpha| + 1, |\alpha| + |\beta|]$.

Primer priponskega drevesa besede »KOKOŠ« je prikazan na Sliki 1. 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 1, znak »$« ne bi bil potreben, ker že znak »Š« jasno določi vse pripone besede »KOKOŠ«. Da zagotovimo, da vse so vse pripone shranjene v listih, se besedi na konc pripne znak »$«.



Slika 1: Primer priponskega drevesa nad besedilom »KOKOŠ$«.

Priponska drevesa so posebna implementacija dreves, zato se lahko nad njimi definira globina vozlišča. Le ta se lahko uporablja pri iskanju in izgradnji drevesa. Globina vozlišča je definirana na sledeči način:

Definicija 2.4. Globina vozlišča $D(v)$ je število vozlišč na poti od korena drevesa do vozlišča v.

Iz Definicije 2.4 je vidno, da sta v globino všteta tudi vozlišče v ter koren drevesa. Ker povezave v priponskih drevesih predstavljajo podnize v besedi T, se lahko nad priponskimi drevesi definira tudi črkovna globina vozliča. Ta je definirana na sledeči način:

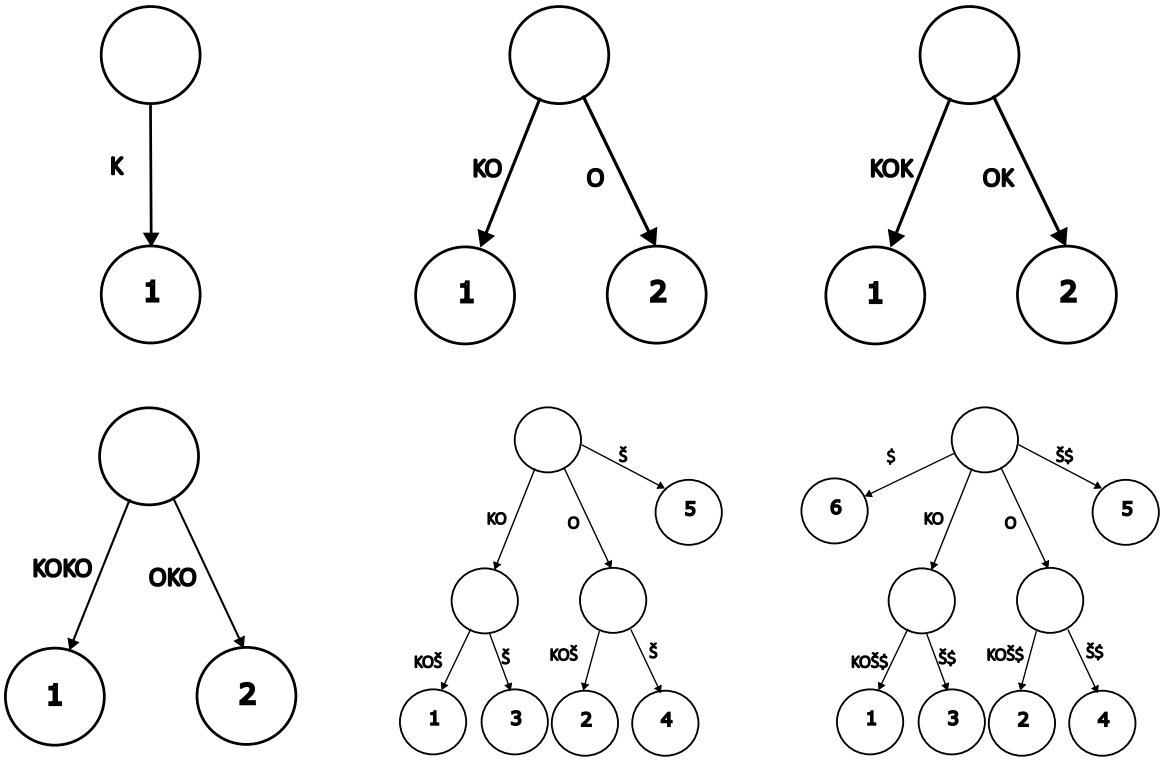
Definicija 2.5. Črkovna globina vozlišča $Sd(v)$ je dolžina podniza pridobljenega s stikom vseh podnizov na povezavah na poti od korena drevesa do vozlišča v.

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

2.1 IZGRADNJA

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

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



Slika 2: Primer izgradnje priponskega drevesa z uporabo Naivne metode za besedo »KOKOŠ$«.

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. Takšnemu drevesu rečemo implicitno priponsko drevo. Ta problem se reši z

beleženjem takih pripon. Če se niz na povezavi, ki predstavlja zadnji del implicitne pripone, nadaljuje po priponi z znakom $T[i]$ se ne stori ničesar, saj je tudi podalšana pripona implicitno predstavljena. Če pa se niz na povezavi, ki predstavlja zadnji del implicitne pripone, ne nadaljuje z znakom $T[i]$, pa se povezavo razdeli na dva dela in se ustavri vozlišče, na katerega kaze povezava z nizom, ki se ujema z predhodnjo pripono, ter ima dva otroka: vozlišče na katerega kaže prostanek predhodnje povezave ter list z znakom $T[i]$. V primeru, da vozlišče že obstaja, se le temu doda novega otroka, na katerega kaže povezava z znakom, ki se razlikuje. Postopek izgradnje priponskega drevesa za besedo »KOKOŠ« z naivno metodo izgradnje je prikazan na Sliki 2. Na Sliki 2 zgolj tretje (zadnje drevo v prvi vrstici) in četrto (prvo drevo v drugi vrstici) drevo sta implicitna priponska drevesa.

Izrek 2.6. *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 $\sum_{j=1}^{i} j = \frac{i(i+1)}{2}$ vozlišč.

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

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

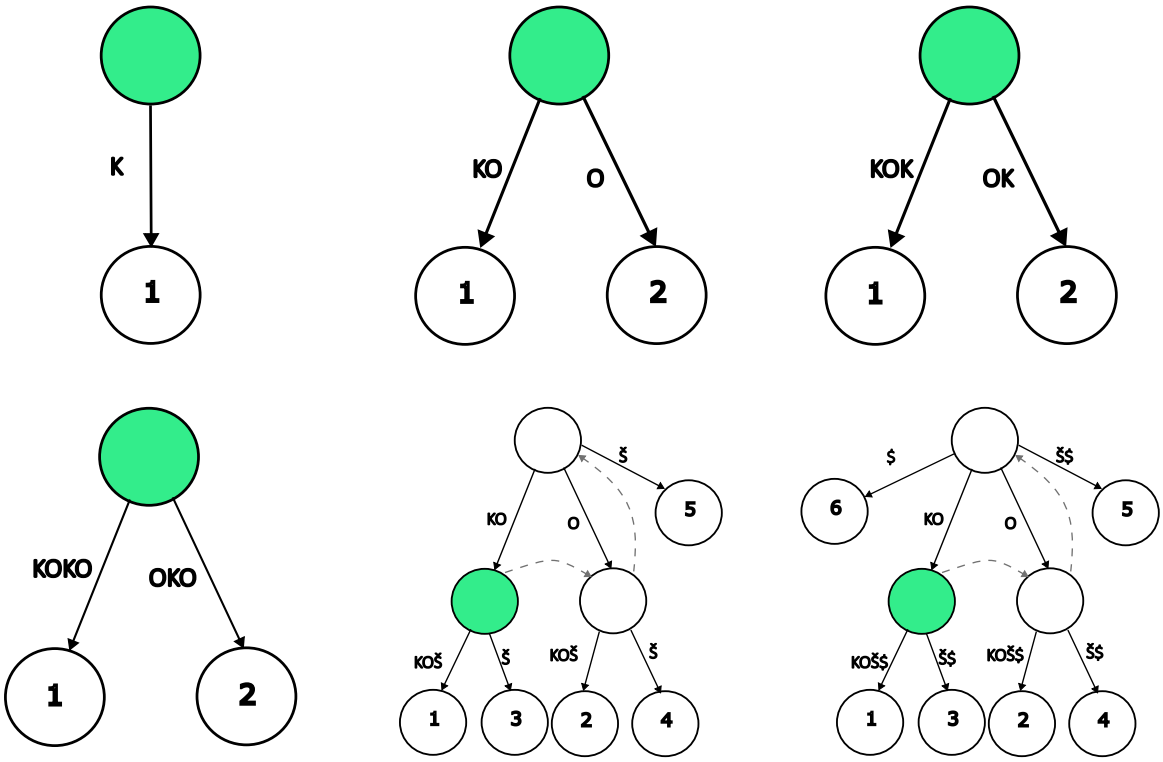
Torej naivna metoda potrebuje $O(n^3)$ časa. □

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

Definicija 2.7. Naj notranje vozlišče v predstavlja niz $x\alpha$, kjer x je znak in $\alpha \in \Sigma^*$ je podniz niza T. Priponska povezava $sl(v)$ je povezava na notranje vozlišče w, ki predstavlja niz α.

Na Sliki 1 sta priponski povezavi prikazani s črtkano puščico. Uvedba priponskih povezav omogoča izogib nepotrebnim sprehodom po drevesu. Tako metoda podaljša vse liste zgolj s sprehodom po priponskih povezavah. Pri tem se opazi, da metoda lahko beleži notranje vozlišče s povezavo na list, ki predstavlja najdaljšo pripono v

predhodnjem koraku izgradnje $T[1, i-1]$. Beleženje tega vozlišča, ki ga imenujemo začetna točka, odstrani nepotrebno iskanje začetka sprehoda po priponskih povezavah.



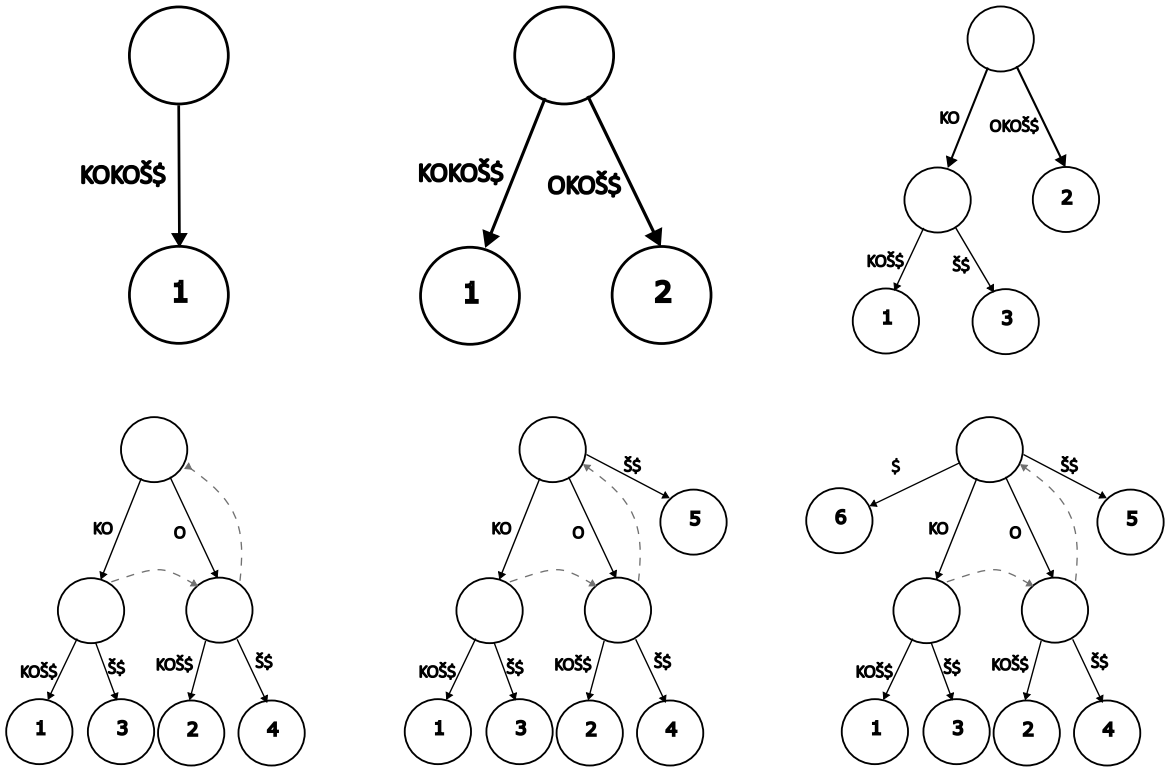
Slika 3: Primer izgradnje priponskega drevesa z uporabo Izboljšane naivne metode za besedo »KOKOŠ$«.

Metoda še vedno potrebje beležiti implicitne pripone (pripone, ki so predpone drugim priponam), ampak to beleženje ne vpliva na čas izgradnje priponskega drevesa. Metoda se v vsakem koraku sprehodi iz začetne točke po priponskih povezavah do korena. Postopek izgradnje priponskega drevesa za besedo »KOKOŠ« z izboljšano naivno metodo izgradnje je prikazan na Sliki 3. Na sliki je začetna točka označena z zeleno barvo. Ker je v i-tem koraku izgradnje največ i eksplicitno definiranih pripon, je tudi število priponskih povezav na poti iz začetne točke do korena po priponskih povezavah enako številu notranjih vozlišč $O(i)$. Torej je čas i-tega koraka $O(i)$. Potemtakem ta metoda potrebuje $O(n^2)$ časa za izgradnjo celotnega drevesa. Iz tega sklepa sledi izrek.

Izrek 2.8. *Izboljšana naivna metoda zgradi priponsko drevo nad besedilom T v času $O(n^2)$.*

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

McCreightov algoritem: McCreightov algoritem ali Algoritem M je prvi algoritem s časovno zahtevnostjo $O(n)$ za izgradnjo drevesa, pri tem pa je tudi prostorska zahtevnost algoritma $O(n)$. Algoritem deluje, tako da se v i-tem koraku v do takrat izgrajeno drevo doda pripona $T[i, n]$. Torej so pripone dodane v drevo v vrstnem redu od najdaljše do najkrajše pripone, za razliko od naivne metode, ki v i-tem koraku izgradnje priponskega drevesa doda i-ti znak v do takrat izgrajeno priponsko drevo, ki je lahko implicitno. Postopek izgradnje priponskega drevesa za besedo »KOKOŠ« z McCreightvim algoritmom je prikazan na Sliki 4.



Slika 4: Primer izgradnje priponskega drevesa z uporabo McCreightvega algoritma za besedo »KOKOŠ$«.

V vsakem koraku je pripona razdeljena na dva dela, in sicer na glavo ter rep. Glava, označena kot $glava_i$, je najdaljša predpona i-te pripone, ki že obstaja v drevesu. Če take pripone ni v drevesu, je glava pripone prazna. Rep besedila, označen kot rep_i, pa je definiran kot preostanek pripone, ki ni del glave. Za razliko od glave, rep ne more biti prazen. Ker algoritem temelji na dejstvu, da nobena pripona ni predpona drugi priponi, se mora n-ti znak besede razlikovati od ostalih znakov v besedi in se ga označi z znakom »$«. Algoritem v vsakem koraku doda krajšo pripono, zato je v repu vedno prisoten vsaj »$«, saj se bo vedno razlikoval od vsakega znaka, ki je na $(n - i)$-tem mestu poljubne že umeščene pripone [4].

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

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

Naslednji korak izgradnje priponskega drevesa je iskanje niza β v drevesu. To imenujemo *rescanning*. Po definiciji $glava_i$ že obstaja v drevesu, torej obstaja tudi pot iz a v c, ki se začne z nizom β. Iskanje poteka tako, da algoritem primerja dolžino $|\beta|$ z dolžino niza na povezavi iz a. Če je $|\beta|$ krajša ali enaka od dolžine niza na povezavi iz a, se iskanje prekine. V primeru, da je $|\beta|$ strogo krajša, se ustvari novo vozlišče d. Če je $|\beta|$ daljša od dolžine niza na povezavi iz a, se izbriše podniz, ki je predstavljen na povezavi, in otrok od a postane novo vozlišče a. Postopek se ponovi, dokler $|\beta|$ ni krajša ali enaka od niza, ki ga predstavlja povezava iz a. V primeru, da ne obstaja priponska povezava iz vozlišča, ki predstavlja niz $x\alpha\beta$, v vozlišče d, se jo ustvari.

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

Izrek 2.9. *McCreightov algoritem zgradi priponsko drevo nad besedilom T v času $O(n)$.*

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

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

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

V i-tem koraku operacija *rescanning* obišče n_i vozlišč. Pri tem se opazi, da v naslednjem koraku ta vozlišča ne bodo obiskana. Iz tega sledi: $|\beta_{i+1}\gamma_{i+1}rep_{i+1}| \leq$

$|\beta_i\gamma_i rep_i| - n_i$, pričemer je $|\beta_n\gamma_n rep_n| = |rep_n| = 1$. Torej velja tudi:

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

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

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

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

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

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

□

Čeprav McCreightov algoritem zgradi priponsko drevo v času in prostoru $O(n)$, algoritem predpostavi, da je beseda T vnaprej poznana. Pri tem se pojavi vprašanje, ali je mogoče izgraditi priponsko drevo, ne da bi vnaprej poznali začetno besedo. Metoda, ki ne potrebuje celotne besede vnaprej, je Ukkonenov algoritem, ki priponsko drevo zgradi sprotno (angl. *on-line*), kot ga je izgradila tudi naivna metoda, ter potrebuje $O(n)$ časa in prostora.

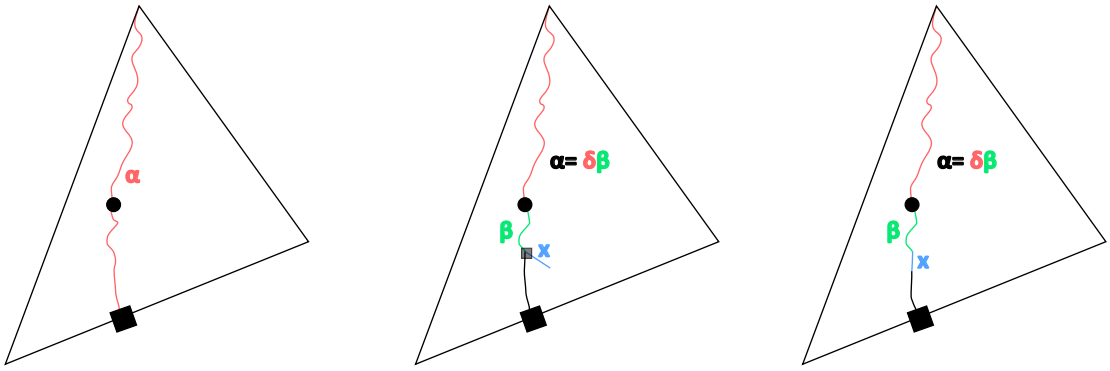
Ukkonenov algoritem: Ukkonenov algoritem deluje na podoben način kot prej predstavljena naivna metoda, saj dodaja v drevo črko po črko. Torej algoritem v i-tem koraku izgradi priponsko drevo, ki je lahko implicitno priponsko drevo, in predstavlja besedilo $T[1, i]$. Primer izgradnje drevesa z Ukkonenovim algoritmom, je prikazan na Sliki 6. Algoritem v i-tem koraku doda v drevo črko x. Naj bo α niz, ki z x tvori niz $T[a, i] = \alpha x$, pri čemer je $1 \leq a < i$ in posledično je α lahko 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 5a.

2. Če pa se niz α ne konča v listu, potem se konča bodisi v vozlišču v, bodisi na povezavi med vozliščema, recimo v_1 in v_2. V tem primeru se lahko znak x doda na dva načina:

(a) Če se nobena pot ne nadaljujeki 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, ki se je ujeml z α. 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 5b.

(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 5c.

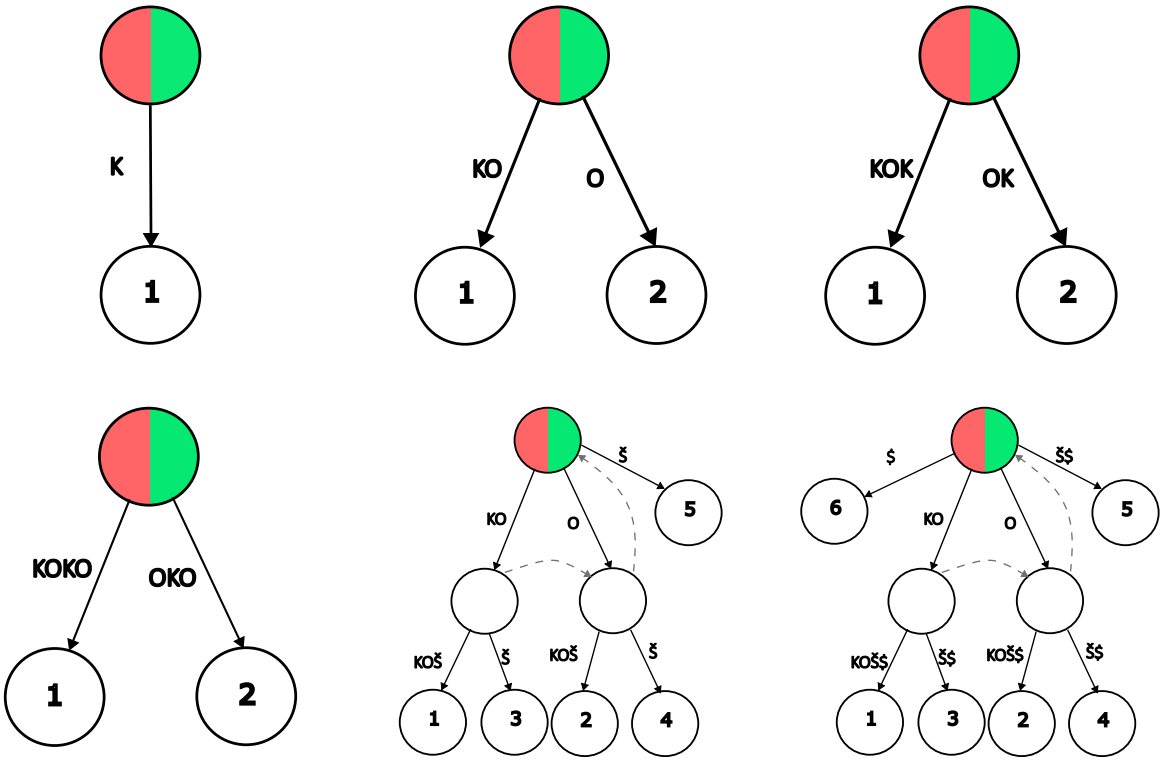


(a) (b) (c)

Slika 5: Načini dodajanja novih znakov v priponsko drevo.

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). Da bo algoritem učinkovit, se hraniti začetek, ki ga imenujemo začetna točka, in konec, ki ga imenujemo končna točka (angl. *end point*), rezčlembe ter dodajanja novih vozlišč v drevesu v trenutnem koraku. Na Sliki 6 je z rdečo označeno vozlišče, ki se nahaja pred končno točko, ter z zeleno je označeno vozlišče, ki pa se nahaja pred začetno točko. Vse točke, v katerih se bo v tem koraku zgodila razčlemba, imenujemo aktivne točke (angl. *active point*). Začetna točka je prva aktivna točka, končna točka pa je zadnja aktivna točka v trenutnem koraku dodajanja. Premik po aktivnih točkah med začetno točko in končno točko poteka po priponskih povezavah. Pri tem pa algoritem v vsakem koraku poti izračuna, ali je že prišel v končno točko. Za to je potrebno hraniti zgolj trenutno aktivno točko ter zastavico, ki hrani vrednost ali je ta točka tudi končna točka. Pri tem velja tudi, da končna točka v koraku $i-1$ postane aktivna točka v koraku i, saj se prva nova razčlemba v koraku i lahko zgodio zgolj v točki, kjer se je končala razčlemba 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 shra-



Slika 6: Primer izgradnje priponskega drevesa z uporabo Ukkonenovaga algoritma za besedo »KOKOŠ$«.

njevanje niza α je le ta shranjen kot par indeksov (k, p), pri čemer je $\alpha = T[k, p]$. Ker je lahko aktivna točka predstavljena z različnimi referenčnimi pari $(s, (k, p))$, se zato lahko za s izbere najglobje vizlišče pred aktivno točko. Takšen par je enoličen in ga imenujemo kanonična oblika referečnega para. 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 6 v četrtem koraku izgradnje, ki ga predstavlja prvo drevo v spodnji vrstici, je začetna aktivna točka predstavljena kot $(koren, (1, 3))$, v naslednjem koraku pa je začetna aktivna točka v $(koren, (1, 4))$, ki se bo razcepila in bo postala novo vozlišče. Ta ista točka je lahko po koncu izgradnje še vedno predstavljena z referenčnim parom $(koren, (1, 4))$, 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 1. V algoritmu s predstavlja najglobje vozlišče pred aktivno točko ter k predstavlja indeks začetne črke niza iz vozlišča s proti aktivni točki v besedilu T. Številka i predstavlja indeks črke v T, ki je dodana v drevo. Vozlišče s ter par števil $(k, i - 1)$ predstavljajo aktivno točko v trenutnem koraku. Zastavica *KončnaTočka* ima vrednost *true*, če je trenutna aktivna

Algoritem 1: Ukkonenov algoritem za izgradnjo priponskega drevesa

Vhod: Besedilo 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 v' 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(sv(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)
```

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 1 uporabi dve pomožni funkciji. Prva uporabljena pomožna funkcija je `kanoničnaOblika`, ki pretvori trenutno aktivno točko priponskega drevesa v kanonično obliko. Funkcija prejme kot vhod vozlišče s ter niz, predstavljen kot položaj začetka k in konca p podniza do aktivne točke v nizu T. 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 znak t, ki se želi vstaviti v priponsko drevo, ter trenutno aktivno točko, ki je podana kot referenčni par $(s, (k, p))$ v kanonični obliki. 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 $(false, v)$.

Izrek 2.10. *Ukkonenov algoritem zgradi priponsko drevo nad besedilom 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 funkcija prever ali če je trenutna aktivna točka tudi končna točka in posledično če ni končna točka jo spremeni v notranje vozlišče. Ker je aktivna točka podana v kanonični obliki, ni potrenbnih odvečnih sprehodov po drevesu, ki bipovečali časovno zahtevnost.

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` obišče $O(n)$ vozlišč skozi celotno izgradnjo priponskega drevesa. Funkcija ob vsakem klicu pogleda največ $p-k$ vozlišč, kar je dolžina niza $\beta = T[k, p]$. Pri tem pa se niz β z vsakim obiskanim vozliščem skrajša, saj se poveča število k. Niz β pa se lahko poveča zgolj v 15 vrstici Algoritma 1. 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 1 se izvede n-krat, medtem ko se zanka v vrstici 6 in funkcija `kanoničnaOblika` vsaka izvede v $O(n)$ časa skozi celotno izgradnjo priponskega drevesa, iz česar sledi, da tudi izgradnja priponskega drevesa potrebuje $O(n)$ časa, da se izvrši.

□

Zaključek: Tako McCreight algoritem [4] kot tudi Ukkonenov algoritem [1] zgradita priponsko drevo v času $O(n)$. Oba algoritma izdelata priponsko drevo, ki ima enake priponske povezave med vozlišči, pri tem pa se zalikujeta v umesnih drevesih. Za primerjavo umesnih dreves se lahko vzame Sliko 4 in Sliko 6, na katerih sta prikazani izgradnji priponskih dreves z obema algoritma za besedo »KOKOŠ$«. V McCreightovem algoritmu vmesno drevo v i-tem koraku predstavlja i najdaljših pripon besedila. Pri čemer pa v Ukkonenovem algoritmu vmesno drevo v i-tem koraku predstavlja priponsko drevo besedila $T[1, i]$, ki je lahko bodisi implicitno bodisi eksplicitno predstavljeno, kar je posledica sprotne izgradnje priponskega drevesa.

Poleg linearne časovne zahtevnosti za izgradnjo priponskega drevesa, oba algoritma potrebujeta $O(n)$ prostora za hrambo in gradnjo priponskega drevesa, saj ima vsako drevo n listov in največ $n-1$ notranjih vozlišč. Vsako vozlišče ima $O(|\Sigma|)$ referenc, pri čemer je 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 4.3, kjer bo tudi izmerjen vpliv pomanjkanja notranjega pomnilnika ter posledična uporaba zunanjega pomnilnika (`Swap` razdelek na zunanjem spominu) namesto notranjega pomnilnika.

3 KOMPAKTNA PREDSTAVITEV

Za daljše besede T priponska drevesa lahko presežejo velikost notranjega pomnilnika. 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), potrebuje približno 144 GB notranjega pomnilnika [13].

Ta problem se da rešiti s kompaktno predstavitvijo podatkovnih struktur, ki se jih bo obravnavalo v tem poglavju. Najprej bo predstavljena podatkovna struktura bitno polje, ki je osnova za ostale kompaktne predstavitve podatkovnih struktur. Za tem bodo prestavljene različne kompaktne predstavitve dreves. Na koncu pa bo predstavljena kompaktna predstavitev priponskega drevesa imenovana kompaktno priponsko drevo (angl. *Compressed Suffix Tree* oziroma CST).

3.1 NOTACIJA IN OSNOVNE OPERACIJE

Bitno polje (angl. *bit vector*) je kompaktna podatkovna struktura, ki je osnova za ostale kompaktne podatkovne strukture. Bitna polja so osnova za kompaktno predstavitev topologijo drevesa. Univerzalna množica vseh bitnih polj dolžine n je velikosti $|N| = 2^n$, torej bitno polje ima prostorsko zahtevnost $n+o(n)$ bitov. Pri tem je potrebnih n bitov za shraniti celotno bitno polje ter dodatnih $o(n)$ bitov za implementacijo operacij nad njim. Bitno polje B podpira tri operacije: $rang_v(B, i)$ (angl. *rank*), $izbira_v(B, i)$ (angl. *select*) ter $dostop(B, i)$ (angl. *access*), ki vrne i-ti bit bitnega polja B.

Definicija 3.1. Operacija $rang_v(B, i)$ vrne število pojav vrednosti $v \in \{1, 0\}$ v bitnem vektorju B do položaja i.

V nadaljevanju bo operacija $rang(B, i)$ predstavljala $rang_1(B, i)$. To je možno, saj velja sledeča relacija med $rang_0(B, i)$ in $rang_1(B, i)$:

$$rang_1(B, i) = i - rang_0(B, i). \tag{3.1}$$

Operacija rang ima časovno zahtevnost $O(1)$, pri tem pa potrebuje dodatno podatkovno strukturo, ki zasede $o(n)$ bitov. V nasprotnem primeru bitno polje ni več kompaktna

podatkovna struktura. Relacija iz enakosti 3.1 omogoča izgradnjo podatkovne strukture zgolj za operacijo $rang_1$.

Pomožna struktura, ki omogoča konstantni čas operacije $rang(B, i)$, shranjuje range različnih elementov bitnega polja B v polju R. Naivni pristop shrani v polju R vrednosti vseh elementov B-ja. Problem tega pristopa je prevelika prostorska zahtevnost, saj R ni več velikosti $o(n)$ bitov. Rešitev tega problema je vzorčenje rangov, tako da v polju R so shranjeni zgolj rangi nekaterih elementov B-ja. Polje R razdeli B na $s = kw$ delov, pri čemer je k poljubno število ter je w dolžina računalniške besede. Element $R[i] = rang_1(B, is)$, pri čemer $0 \leq i \leq \lfloor \frac{n}{s} \rfloor$. Na ta način je operacija rang implementiran kot

$$rang_1(B, i) = R\left[\left\lfloor \frac{i}{s} \right\rfloor\right] + \texttt{popcount}\left(B\left[\left\lfloor \frac{i}{s} \right\rfloor s, i\right]\right), \tag{3.2}$$

kjer je `popcount` funkcija, ki prešteje število bitov z vrednostjo 1. Zaradi uporabnosti te funkcije je le ta implementirana v različnih modernih arhitekturah procesorjev. Polje R ima velikost $\lfloor \frac{n}{k} \rfloor$ bitov, saj shranjuje $\lfloor \frac{n}{s} \rfloor$ števil dolžine w. Pri tem je potrebno funkcijo `popcount` pognati največ k krat, kar naredi čas operacije rang $O(k)$ [6].

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

$$rang_1(B, i) = R\left[\left\lfloor \frac{i}{kw} \right\rfloor\right] + R'\left[\left\lfloor \frac{i}{w} \right\rfloor\right] + \texttt{popcount}\left(B\left[\left\lfloor \frac{i}{w} \right\rfloor, \left\lfloor \frac{i}{w} \right\rfloor + (i \mod w)\right]\right). \tag{3.3}$$

V praksi se izkaže, da se najboljše rezultate pridobi, ko je $k = 8$ za računalniško besedo dolžine $w = 64$ (uporabljena v vseh modernih procesorjih). To omogoča shranjevanje vedra v R in vseh pripadajočih veder v R' v dveh besedah, ker prva beseda shrani vrednost vedra v R druga pa vse vrednosti pripadajočih veder v R'. Ta implementacija potrebuje $1,25 * n$ bitov za vse tri podatkovne strukture skupaj [6].

Podobno kot operacija rang, tudi operacija izbira potrebuje pomožno podatkovno strukturo za izvedbo v konstantnem času.

Definicija 3.2. Operacija $izbira_v(B, i)$ vrne položaj i-tega pojava vrednosti $v \in \{1, 0\}$ v bitnem vektorju B.

Operacijo izbira si lahko predstavimo, kot inverzno operacijo od operacije rang, saj velja relacija $j = rang_v(B, izbira_v(B, j))$. Pri tem pa ne obstaja povezava med operacijo $izbira_1(B, i)$ in $izbira_0(B, i)$. To pomeni, da rešitev z pomožno podatkovno strukturo za $izbira_1(B, i)$ ne more biti uporabljena pri iskanju rešitve za $izbira_0(B, i)$.

V nadaljevanju bo opisan postopek iskanja $izbire_1(B, i)$, saj se lahko $izbira_0(B, i)$ implementira na podoben način [6].

Ko operacija izbira ni ključna pri reševanju problemov, se lahko uporabi binarno iskanje v poljih R in R', ki potrebuje $O(\log n)$ 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 $O(\log \log n)$ z uporabo pomožnih podatkovnih struktur. Podobno kot pri operacij rang se bitno polje razdeli na $\lceil \frac{m}{s} \rceil$ veder, pri čemer m je število bitov z vrednostjo 1 v bitnem polju in s je število takih bitov v vsakem vedru. Polje $S[0, \lceil \frac{m}{s} \rceil]$ hrani vrednosti $S[i] = izbira_1(B, i * s + 1)$, pri čemer $S[\lceil \frac{m}{s} \rceil] = n + 1$. Polje S potrebuje $w(\lceil \frac{m}{s} \rceil + 1)$ bitov. Ko je $s = w^2$, potem podatkovna struktura potrebuje $\lceil \frac{m}{w^2} \rceil = o(m) = o(n)$ bitov [6].

Vedra niso enako velika, zato se lahko razdelijo na velika vedra in mala vedra, pri čemer je vedro veliko natanko tedaj, ko je večje kot $s = \log^2 n$ bitov, sicer je malo vedro. Pri tem je potrebno shraniti velikost vedra v bitno polje V, kjer $V[i] = 1$, če i-to vedro je veliko, sicer $V[i] = 0$. Bitno polje V potrebuje tudi dodatno podatkovno strukturo za rang. Za velika vedra se izračuna vseh s vrednosti izbire, pri čemer so shranjeni v polju I položaji bitov z vrednostjo ena znotraj vedra. Za mala vedra, pa se sproti naračuna izbira znotraj vedra. Operacija izbira se izračuna na sledeči način:

$$izbira_1(B, j) = \begin{cases} S\left[\left\lceil \frac{j}{s} - 1 \right\rceil\right] + I\left[rang_1(V, \lceil \frac{j}{s} \rceil)s + x\right], & V\left[\left\lceil \frac{j}{s} \right\rceil\right] = 1 \\ S\left[\left\lceil \frac{j}{s} - 1 \right\rceil\right] + k, & V\left[\left\lceil \frac{j}{s} \right\rceil\right] = 0 \\ n + 1, & m < j \end{cases}, \tag{3.4}$$

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

Polje I mora shraniti $s\lceil \log n \rceil$ bitov za vsako veliko vedro, katerih je $\frac{n}{s(\log n)^2}$, torej potrebuje $O\left(\left\lceil \frac{n}{\log n} \right\rceil\right) = o(n)$ 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 $o(n)$ bitov [6].

Operacija se lahko izvedene 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 $s' = (\log \log n)^2$ mini veder, pri čemer mini vedro je veliko, ko je večje od $s'(\log \log n)^2$. Vsako majhno mini vedro potrebuje $s' \log\left(s(\log n)^2\right) = O((\log \log n)^3) = o(n)$ bitov ter vsako veliko mini vedro potrebuje isto prostora, pri čemer pa jih je največ $O\left(\frac{n}{(\log \log n)^4}\right)$, torej vsa velika mini vedra skupaj potrebujejo $O\left(\frac{n}{\log \log n}\right) = o(n)$ bitov [6].

Vse dodatne podatkovne 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 $O(n)$ časa. Pri tem je ves potrebni spomin že predhodno dodeljen.

3.2 KOMPAKTNA PREDSTAVITEV DREVES

Z uporabo bitnih polj je mogoče predstaviti mnogo podatkovnih struktur. Primer uporabe bitnih polj je kompaktna predstavitev topologije dreves. Običajno je podatkovna struktura drevo sestavljeno iz vozlišč ter kazalcev (angl. *pointers*). Vsako vozlišče shrani lastno vrednost ter kazalce na svoje otroke. Drevo z n-timi vozlišči potrebuje $O(n \log n)$ bitov za shraniti topologijo drevesa. Kompaktna predstavitev topologije dreves zniža prostorsko zahtevnost na $2n + o(n)$ bitov. Za vsako vozlišče je potrebno predstaviti kazalec na vozlišče ter predstaviti, da je vozlišče bilo že obiskano, torej je potrebnih $2n$ bitov.

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

Definicija 3.3. Topologija podatkovne strukture drevo mora podpirati naslednje operacije:

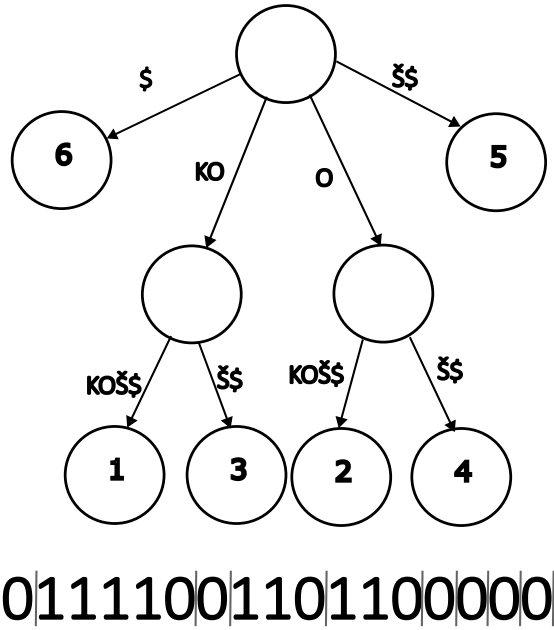
1. $koren()$: vrne koren priponskega drevesa,
2. $jeList(v)$: vrne »Da«, če je vozlišče list, sicer vrne »Ne«,
3. $stOtrok(v)$: vrne število otrok vozlišča v,
4. $otrok(v, i)$: vrne vozlišče w, ki je i-ti otrok vozlišča v,
5. $prviOtrok(v)$: vrne vozlišče w, ki je prvi otrok vozlišča v,
6. $nbrat(v)$: vrne vozlišče w, ki je naslednji brat od vozlišča v,
7. $pbrat(v)$: vrne vozlišče w, ki je predhodni brat od vozlišča v,
8. $starš(v)$: vrne vozlišče w, ki je starš od vozlišča v,
9. $globina(v)$ vrne število vozlišč na poti iz korena do vozlišča v,
10. $lca(v, w)$: vrne najnižjega skupnega prednika od v in w.

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

3.2.1 Zaporedje eniških zapisov stopenj vozlišč po plasteh

Prvi način zapisa topologije drevesa je zaporedje eniških zapisov stopenj vozlišč po plasteh (LOUDS). Drevo se predstavi kot bitno polje B dolžine $2n + 1$, pri čemer je n število vozlišč v drevesu. Zapis drevesa se začne z nizom 10, temu pa sledi zapis vsakega vozlišča. Vsako vozlišče v je predstavljeno z nizom 1^o0, pri je o predstavlja število otrok posameznega vozlišča, torej se ena zapiše o-krat in nato eno ničlo, ki predstavlja konec vozlišča. Kot je razvidno iz imena predstavitve, so vozlišča obiskana po nivojih: vozlišča z isto globino so zaporedno predstavljana [6].

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



Slika 7: Primer predstavitve drevesa z metodo LOUDS za priponsko drevo besede »KOKOŠ$«.

Med izgradnjo kompaktne predstavitve se vsa vozlišča indeksira s številom 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 s pomočjo predstavitve drevesa LOUDS. Vrednost x predstavlja indeks vozlišča in se izračuna kot $x = izbira_0(B, v) + 1$ [6].

V Tabeli 1 sta uporabljeni dve novi operaciji nad bitnimi polji: $naslednik_v(B, y)$ in $predhodnik_v(B, y)$, kjer je $v \in \{1, 0\}$. Operacija predhodnik elementa y najde element x_i del zaporedja, za katerega velja $x_i \leq y < x_{i+1}$ in $1 \leq i \leq m$, kjer je m število

Tabela 1: Implementacija operacij drevesa v LOUDS.

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

ponovitev v-ja v B. Operacija je implementirana kot

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

pri čemer je $v \in \{1, 0\}$. Na podoben način je definirana tudi operacija naslednik. Operacija naslednik elementa y najde položaj elementa x_i v zaporedju $x_1 \leq x_2 \leq \cdots \leq x_m$, pri čemer velja, da $x_{i-1} < y \leq x_i$. Pri tem je implementirana kot

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

kjer je $v \in \{1, 0\}$. Zaporedje $x_1 \leq x_2 \leq \cdots \leq x_n$ predstavlja vozlišča v zaporedju eniških zapisov stopenj vozlišč po plasteh. Torej operaciji predhodnik in naslednik vrneta začetni položaj predhodnega oziroma naslednjega vozlišča v bitnem polju B. Primer uporabe je operacija $stOtrok(v)$, ki izračuna število otrok, tako da odšteje položaj vozlišča v v B od položaja naslednjega bita z vrednostjo 0, ki predstavlja vozlišče v. Vse operacije razen operacije *globina* in *lca* so izvršene v konstantnem času, z uporabo dodatne podatkovne strukture za rang in izbiro. Pri tem je treba izgraditi podatkovno strukturo zgolj za $izbira_0$, saj operacija $izbira_1$ ni potrebna za pravilno delovanje operacij drevesa [6].

Operacija *globina* ni podprta, saj LOUDS predstavitev ne omogoča učinkovitega iskanja. Pri tem pa velja, da $globina(u) \geq globina(v)$, če je $u > v$. S pomočjo tega dejstva se lahko implementira operacijo *lca*, kot je prikazano v Algoritmu 2 [6].

3.2.2 Uravnoteženi oklepaji

Naslednji 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

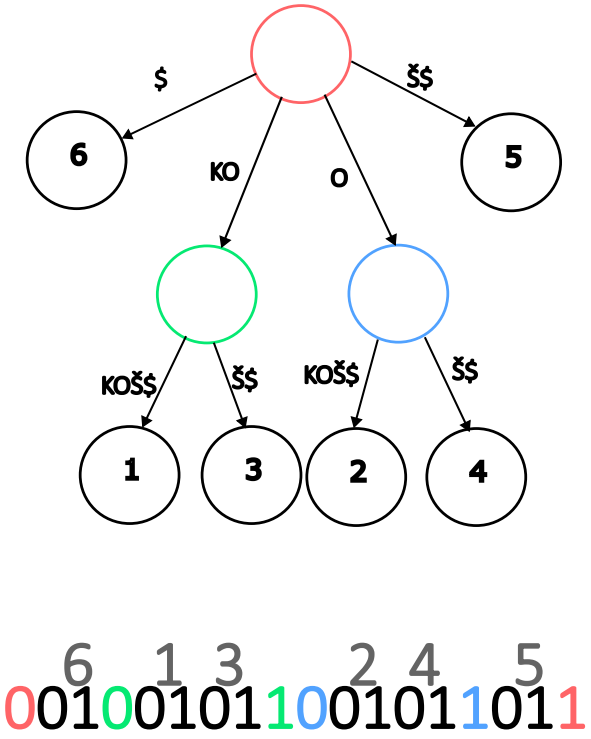
Algoritem 2: Operacija $lca(v, w)$ (LOUDS)

Vhod: Bitno polje B, vozlišča v in w

Izhod: Vozlišče u

```
1 while v ≠ w do
2     če v > w potem
3         v<-staršB, v)
4     sicer
5         w<-staršB, w)
6 vrni v
```

v drevesu. Zaporedje se zgradi s pomočjo 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 zapiše s številom 1. Primer priponskega drevesa predstavljenega z uporabo zaporedja uravnoteženih oklepajev je prikazan na Sliki 8. 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].



Slika 8: Primer predstavitve drevesa z metodo BP za priponsko drevo besede »KOKOŠ$«.

Torej je vsako vozlišče je predstavljeno kot par '(' in ')'. Tako je mogoče definirati sledeče operacije nad oklepaji: *odpri*, *zapri*, *višek* ter *oklepa*. Operacija $višek(B, i)$ vrne število '(', ki niso bili še zaprti in je

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

Operacija $odpri(B, i)$ vrne položaj '(', ki odpre ')' na i-tem mestu v B. Operacijo $odpri(B, i)$ se lahko definira tudi, tako da vrne prvi $j > i$, pri čemer $višek(B, j) = višek(B, i) - 1$. Podobno operacija $zapri(B, i)$ vrne položaj ')', ki zapre '(' na i-tem mestu v B. Z uporabo operacije *višek* je operacija *zapri* definirana, tako da vrne največji $j < i$, pri čemer je $višek(B, j) = višek(B, i) + 1$. Operacija $oklepa(B, i)$ vrne položaj j od '(' v bitem polju B, pri čemer velja, da $j < i < zapri(B, j)$. Z uporabo operacije *višek* operacija $oklepa(B, i)$ vrne položaj največjega $j < i$, pri čemer je $višek(B, j - 1) = višek(B, i) - 2$ [6].

Pravilno delovanje drevesa, ki je predstavljen z uporabo BP, zahteva še operacije o višku nad odseki bitnega polja B. Te operacije so sledeče: rmq, rMq, *minizbira* in *minštetje*. Operacija $rmq(B, i, j)$ (angl. *range minimum query*) vrne položaj k, kjer je $i \leq k \leq j$, $višek(B, k)$ je najnižji višek v območju med i in j ter k je prvi pojav najnižjega viška na tem območju. Podobno je definirana tudi operacija $rMq(B, i, j)$ (angl. *range maximum query*), ki pa vrne položaj k, pri čemer je $i \leq k \leq j$ in $višek(B, k)$ je najvišji višek v območju med i in j ter k je prvi pojav najvišjega viška na območju. Operacija $minizbira(B, i, j, t)$ vrne položaj t-tega pojava najmanjšega viška na območju med i in j. Operacija $minštetje(B, i, j)$ pa vrne število pojav najnižjega viška na območju med i in j [6].

Vse predstavljene operacije (razen operacije *višek*, ki potrebuje $O(1)$ časa za izvajanje) se lahko implementira s časovno zahtevnostjo $O(\log n)$. Pri tem pa je potrebno zgraditi dodatno podatkovno strukturo rmM-drevo (angl. *range minumum maximum tree*), ki je lahko shranjeno z $O(n/\log n) = o(n)$ dodatnimi biti. Podatkovna struktura razdeli bitno polje B na $\frac{n}{b}$ veder velikosti b ter za vsako vedro se ustvari list rmM-drevesa. Drevo je levo poravnano in se lahko zapiše kot polje vozlišč (podobno kot podatkovna struktura kopica). Torej so otroci i-tega vozlišča na $2i$-tem in $2i + 1$-vem mestu v polju ter starš i-tega vozlišča se nahaja na $\lfloor \frac{i}{2} \rfloor$-mestu. Vsako vozlišče v drevesu hrani štiri podatke: e razlika viška med začetkom in koncem pokritega območja, m najmanjši relativni višek v območju, M največji relativni višek na območju in n število najmanjših viškov na območju, ki ga pokriva vozlišče. Pri tem vozlišče pokriva celotno območje, ki ga pokrivata oba otroka, in listi pokrivajo zgolj eno vedro dolžine b. Torej koren drevesa pokriva celotno bitno polje B. Vse predstavljene operacije so implementirane s pomočjo sprehoda po rmM-drevesu [6].

Podobno kot pri predstavitvi drevesa LOUDS, tudi predstavitev BP potrebuje dodatno podatkovno strukturo za operaciji *izbira* in *rang*. Pri tem sta potrebni zgolj

podatkovni strukturi za 0, saj operaciji $rang_1$ in $izbira_1$ nista potrebni za pravilno delovanje drevesa v tej predstavitvi. Podatkovni strukturi potrebujeta vsaka $o(n)$ dodatnih bitov in operaciji se izvršita v konstantnem času.

Tabela 2: Implementacija operacij drevesa z BP

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

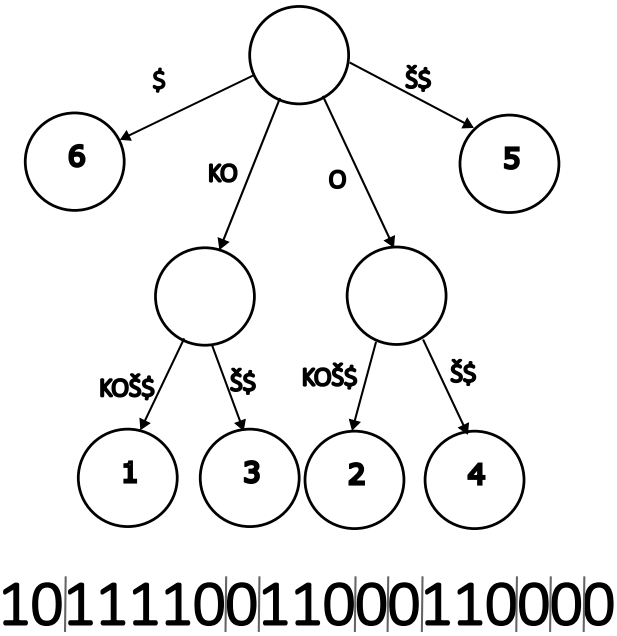
V Tabeli 2 so prikazane implementacije operacij, ki so potrebne za pravilno delovanje drevesa. Indeks $x = izbira_0(B, v)$ je uporabljen za pridobiti dodatne informacije o vozlišču, saj vrednost v predstavljata položaja '(' v zaporedju B, ne pa indeksa v tabeli z dodatnimi informacijami o vozlišču.

Zapis topologije drevesa s BP omogoča dodatne operacije. Primer operacije je oštevilčenje in iskanja listov drevesa, kar je storjeno s pomočjo posplošitve operacije rang in izbiro na poljubno dolge nize. Ker imajo listi imajo 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.

3.2.3 Zaporedje eniških zapisov stopenj vozlišč v globino

Zadnja predstavitev topologije drevesa je zaporedje eniških zapisov stopenj vozlišč v globino (DFUDS). Predstavitev omogoča preprostejše implementacije operacij ter posledično manjše dodatne podatkovne strukture, za razliko od zapisa z zaporedjem uravnoteženih oklepajev. Pri tem pa omogoča hitrejše implementacije nekaterih operacij v primerjavi, z uporabo zaporedja eniških zapisov stopenj vozlišč po plasteh. Primer zapisa topologije drevesa z DFUDS je prikazan na Sliki 9. Vozlišča so v zaporedju ločena s sivimi črtami.

Drevo je predstavljeno z bitnim poljem $B[1, 2n + 2]$. Zapis temelji na uporabi zaporedja uravnoteženih oklepajev in zapisu stopenj vozlišč, zato je potrebno dodati



Slika 9: Primer predstavitve drevesa z metodo DFUDS za priponsko drevo besede »KOKOŠ$«.

na začetek bitnega polja $B[1,3] = 110$. Zatem pa so zapisane stopnje vozlišč, kot zaporedje 1^o0 pri čemer o je število otrok vozlišča. Zapis je zgrajen z uporab pregleda v globino. Vsa vozlišča imajo indeks, ki je zaporedno število obiskanega vozlišča.

Tabela 3: Implementacija operacij drevesa z DFUDS

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

V Tabeli 3 so predstavljene implementacije operaciji nad drevesom implementirane z DFUDS. Indeks $x = izbira_0(B, v)+1$ je uporabljen za shranjevanje dodatnih informacij o vozlišču, saj je vozlišče v predstavljeno s položajem vozlišča v bitnem polju B. V tabeli je vidno, da operacija $globina(v)$ ni podprta, saj ni mogoče implementirati te operacije brez sprehoda od vozlišča v do korena ter pri tem šteti potrebne korake.

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

Pri implementaciji operacije $nbrat(v)$ pa je prisotna operacija $iskanjeNaprej(B, i, d)$. Operacija omogoča iskanje prvega pojava viška, ki je za d večji od viška na i-tem mestu v bitnem polju B. Operacija je definirana kot

$$iskanjeNaprej(B, i, d) = \min\{j > i; višek(B, j) = višek(B, i) + d\} \cup \{n + 1\}.$$

Operacija $iskanjeNaprej$ je lahko implementirana s pomočjo uporabe rmM-drevesa in potrebuje $O(\log n)$ časa, da se izvrši. Implementirana je s pomočjo sprehoda po rmM-drevesu in uporabo viška na intervalu pokritega z vozliščem v ($rmM[v].e$, pri čemer je polje rmM predstavitev rmM-drevesa) in najmanjšega relativnega viška na intervalu pokritim z vozliščem v ($rmM[v].m$, pri čemer polje rmM je predstavitev rmM-drevesa).

Vse operacije, ki temeljijo na operaciji $rang$ in $izbira$, se izvršijo v $O(1)$ času. Operacije, ki pa 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. Pri tem je mogoče zmanjšati prostorsko zahtevnost rmM-drevesa, tako da se shranita zgolj polji e in m. Tako se razpolovi prostorsko zahtevnost drevesa, ki pa še vedno zahteva $o(n)$ dodatnih bitov, brez škode za implementacijo operacij drevesa. Tega ni mogoče storiti v BP, saj $rMq(B, i, j)$ se uporablja pri implementaciji dodatnih operaciji, primer take operacije je $najgloljeVozlišče(v)$, ki vrne najgloljše vozlišče v poddrevesu s korenom v.

Podobno kot pri predstavitvi drevesa z uporabo zaporedja uravnoteženih oklepajev je mogoče implementirati dodatne operacije nad listi. Za razliko od uravnoteženih oklepajev, kjer ima list obliko 10, ima list obliko 00 v zaporedju eniških stopenj v globino. Prva 0 predstavlja konec predhodnega vozlišča. Tako je možno implementirati $rang_{00}$ in $izbira_{00}$, ki omogočata iskanje listov v drevesu. Pri tem sta potrebi popravljeni podatkovni strukturi za $rang$ in $izbiro$, ki pa potrebujeta vsaka $o(n)$ dodatnih bitov ter še vedno omogočata konstanten čas izvajanja posplošenih operacij $rang_{00}$ in $izbira_{00}$.

3.3 KOMPAKTNA PRIPONSKA DREVESA

Predstavljene topologije dreves bodo uporabljene za izdelavo podatkovne strukture, ki je ekvivalentna Priponskemu drevesu. Nova podatkovna struktura se imenuje Kompaktno priponsko drevo (angl. *Compressed Suffix Trees* oziroma CST).

Sledeča definicija Sadakane [2] definira abstraktno podatkovno strukturo priponsko drevo in poda vse operacije, ki so potrebne za pravilno delovanje priponskega drevesa.

Definicija 3.4. Abstraktna podatkovna struktura Priponsko drevo nad besedilo T podpira sledeče operacije:

1. $koren()$: vrne koren priponskega drevesa,
2. $jeList(v)$: vrne »Da«, če je vozlišče list, sicer vrne »Ne«,
3. $otrok(v, z)$: vrne otroka w, katerega povezava se začne z znakom z. Če otrok ne obstaja vrne 0,
4. $prviOtrok(v)$: vrne vozlišče w, ki je prvi otrok vozlišča v,
5. $nbrat(v)$: vrne vozlišče w, ki je naslednji brat od vozlišča v,
6. $pbrat(v)$: vrne vozlišče w, ki je predhodni brat od vozlišča v,
7. $starš(v)$: vrne vozlišče w, ki je starš od vozlišča v,
8. $povezava(v, i)$: vrne i-to črko na povezavi do vozlišča v,
9. $globinaNiza(v)$ vrne število znakov na poti iz korena do vozlišča v,
10. $lca(v, w)$: vrne najnižjega skupnega prednika od v in w,
11. $sl(v)$: vrne vozlišče w, na katerega kaže priponska povezava iz vozlišča v.

Kasai idr. [14] so predstavili način simulacije priponskega drevesa s pomočjo priponskega polja (angl. *Suffix Array* oziroma SA) in polja najdaljših skupnih predpon (angl. *Longest Common Preffix* oziroma LCP). Z uporabo kompaktnih različic teh dveh podatkovnih struktur je možno implementirati kompaktno priponsko drevo. Veliko operacij iz Definicije 3.4 so operacije nad drevesi, zato je potrebno dodati še kompaktno predstavitev topologije drevesa, saj le ta pospeši operacije nad drevesi. Torej sledi definicija za podatkovno strukturo kompaktno priponsko drevo.

Definicija 3.5. Kompaktno priponsko drevo nad besedilom T je sestavljeno iz sledečih podatkovnih struktur:

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

V nadaljevanju poglavja bo predstavljena Sadakanejeva [2] implementacija kompaktnega priponskega drevesa. V Poglavju 3.2 je bilo predstavljenih več različnih predstavitev topologije dreves, implementacija kompaktnega priponskega drevesa pa uporablja predstavitev z zaporedjem uravnoteženih oklepajev.

Topologija drevesa mora podpirati operacije $rang_p$, $izbira_p$, $zapri$, $odpri$ in $oklepa$, ki se izvajajo v konstantnem času za $p \in \{0, 1\}^*$. Pri tem so potrebne dve dodatni podatkovni strukturi za operacijo $rang$ ($rang_0$, $rang_{01}$) in tri za operacijo $izbira$

($izbira_0$, $izbira_1$ in $izbira_{01}$). Ostale operacije pa uporabljajo podatkovno strukturo za višek predstavljeno v [15], ki je implementirana na podoben način kot dodatna podatkovna struktura za *rang*, namesto rmM-drevesa. To pomeni, da za operacijo $otrok(v, i)$ je potrebno $O(|\Sigma|)$ časa. Zato tudi operacija $lca(v, w)$ potrebuje dodatno podatkovno strukturo, ki ji omogoča konstanten čas izvajanja. Ta podatkovna struktura potrebuje $o(n)$ dodatnih bitov. Dodatna podatkovna struktura L za operacijo $lca(v, w)$ deluje na podoben način kot podatkovna struktura za *rang*, pri čemer $L[i]$ je višek na vzorčnem mestu i. Za hitrejše delovanje operacije se zgradi še dvodimenzionalno tabelo $M[i, k]$, ki shrani položaj najmanjšega viška na intervalu $L[i..i + 2^k - 1]$. Vrednost k je manjša od velikosti vedra. Torej operacija $lca(v, w)$ se izračuna kot $\min\{M[v', k], M[w' - 2^k + 1, k]\}$, pri čemer je $k = \lfloor \log(w' - v') \rfloor$, v' predstavlja vedro v L, ki vsebuje v, ter w' je vedro v L, ki vsebuje w.

V primeru, da so potrebne dodatne operacije, ki temeljijo na rMq operaciji in $|\Sigma| > \log n$, se lahko doda še rmM-drevo, ki omogoča izvedbo operaciji v $O(\log n)$ ter pospeši izvajanje oziroma omogoča implementacijo tudi drugih operacij.

Lema 3.6. *Podatkovna struktura za predstavitev topologije priponskega drevesa τ nad besedilom T dolžine n potrebuje $4n + o(n)$ bitov.*

Dokaz. Priponsko drevo nad besedilom T ima $2n - 1$ vozlišč: od tega je $n - 1$ notranjih vozlišč in n listov. Zato je potrebnih $4n - 2$ bitov za zapis topologije drevesa τ s sekvenco uravnoteženih oklepajev. Torej celotna podatkovna struktura topologije drevesa potrebuje $4n + o(n)$ bitov, saj vsaka dodatna podatkovna struktura potrebuje še dodatnih $o(n)$ bitov. □

Naslednja podatkovna struktura, ki je potrebna za pravilno delovanje kompaktnega priponskega drevesa, je kompaktno priponsko polje (angl. *Compressed Suffix Array* oziroma CSA). Priponsko polje (angl. *Suffix Array* oziroma SA) je podatkovna struktura, v kateri so shranjene vse pripone besedila T v leksikografskem vrstnem redu. Priponska polja so implementirana kot polje indeksov začetkov pripon (cela števila), torej potrebujejo $O(n \log n)$ bitov (v praksi $O(nw)$ bitov, saj implementacije uporabljajo prevzeto dolžino celega števila). Kompaktna priponska polja znižajo prostorsko zahtevnost 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 imenovan FM-indeks pa uporablja LF funkciji, ki je inverzna funkcija od Ψ [2, 6].

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

Definicija 3.7. Kompaktno priponsko polje nad besedilom T, ki je uporabljeno za predstavitev kompaktnega priponskega drevesa, podpira sledeče operacije:

1. $pripona(i)$ vrne $SA[i]$ v času t_{SA},
2. $inverz(i)$ vrne $j = SA^{-1}[i]$, pri čemer je $SA[j] = i$, v času t_{SA},
3. $\Psi(i)$ vrne $SA^{-1}[SA[i] + 1]$ v času t_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 konstantnem času oziroma v času t_Ψ, čeprav ni potrebna za pravilno delovanje kompaktnega priponskega polja. Iz definicije funkcije Ψ je razvidno, da jo lahko simuliramo z uporabo operacij $inverz(i)$ in $pripona(i)$. Iz definicije 3.7 sledi, da mora kompaktno priponsko polje podpirati funkcijo Ψ, zato implementacije z FM-indeksom ne pridejo v poštev.

Vse štiri operacije so lahko 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 polja Ψ temelji na dejstvu, da za $j = SA[i]$ in $j' = SA[i+i]$ ter $T[j] = T[j']$ sledi $T[j+1, n] < T[j'+1, n]$ (leksikografsko), torej tudi $\Psi(i) < \Psi(i + 1)$.

Dejstvo se lahko zapiše: kot bitno polje D, pri čemer $D[i] = 1$, ko je $i = 1$ ali $T[SA[i]] \neq T[SA[i + 1]]$, ter polje znakov S urejenih v leksikografskem vrstnem redu, tako da $T[SA[i]] = S[rang(D, i)]$.

Polje Ψ se lahko razdeli na $|\Sigma|$ delov in uvede polje C, za katerega velja $C[c] = i$, tako da $T[SA[i + 1]] = c$ in $T[SA[i]] \neq c$, za vsak $c \in \Sigma$. Polje C potrebuje $O(|\Sigma| \log n)$ bitov. Tako se lahko razdeli Ψ na 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 $c = rang_1(D, i)$. Bitna polja B_c so zelo redka (število enic je bistveno manjše kot število ničel), torej se jih lahko stisne iz $|\Sigma|n + o(n)$ bitov na $n_c \log \frac{n}{n_c} + O(n_c)$ bitov, kar pomeni, da funkcija Ψ potrebuje $nH_0(T) + O(n + |\Sigma|w)$ bitov, pri čemer $H_0(T)$ je entropija besedila T [6].

Do sedaj predstavljena podatkovna struktura omogoča zgolj iskanje števila ponovitev vzorca v besedilu, ne pa lokacije pojavov vzorca v besedilu. Za to sta potrebni dodatni podatkovni strukturi, ki nadomestita priponsko polje SA ter inverzno polje SA^{-1}. Priponsko polje SA se vzorči $l = \Theta(\log n)$-krat. Pri tem se uporabi dodatno bitno polje B, 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 oziroma $i_k = \Psi^k[i]$ (na primer $i_2 = \Psi^2[i] = \Psi[\Psi[i]]$) [6].

Podobno se vzorči tudi polje inverzov pripon SA^{-1} le da se ta vzorči na enakomernih intervalih dolžine l. Polje vzorcev $SA_S^{-1}[1 : \lfloor n/l \rfloor]$ se uporabi za izračun vrednoti $SA^{-}1[i]$. 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 $\Psi^{i-i'}[SA_S^{-1}[i'/l]]$ [6].

Tako predstavljeno kompaktno priponsko polje potrebuje $nH_0(T) + O(n + |\Sigma|w)$ bitov za implementacijo. Potrebni čas za izračun funkcije Ψ je $t_\Psi = O(1)$ ter potrebni čas za iskanje po priponskem polju in inverznem priponskem polju je $t_{SA} = t_\Psi \log n = O(\log n)$.

Zadnja podatkovna struktura, ki sestavlja kompaktno priponsko drevo, je polje najdaljših skupnih predpon (angl. *Longest Common Prefixs* oziroma LCP). V polju so shranjene dolžine najdaljše predpone dveh zaporednih pripon. Polje $LCP[2, n]$ dolžine $n - 1$ je sestavljeno iz celic z vrednostjo

$$LCP[i] = lcp(SA[i - 1], SA[i]);\ 2 \leq i \leq n,$$

pri čemer je funkcija $lcp(X, Y) = \max\{i; X[1, i] = Y[1, i]\}$. Polje LCP je lažje shraniti kot permutacijo $PLCP$, pri čemer $PLCP[i] = LCP[SA^{-1}[i]]$ ali $LCP[i] = PLCP[SA[i]]$ [6].

Permutacijo $PLCP[1, n - 1]$ je lažje shraniti, saj je zaporedje $PLCP[i] + 2i$ za vsak i med 1 in $n - 1$ strogo naraščajoče. Saj je $PLCP[i + 1] \geq PLCP[i] - 1$. To enostavno dokažemo, saj obstajata $T[j, n]$ in $T[i, n]$, $T[i, n] < T[j, n]$, ki imata najdaljšo skupno predpono dolžine $PLCP[i] > 0$. Potem ima $T[i + 1, n]$ najdaljšo skupno predpono s $T[j + 1, n]$ dolžine $PLCP[i] - 1$ in vsi nizi, ki so leksikografsko med $T[i + 1, n]$ in $T[j + 1, n]$, imajo najdaljšo skupno predpono s $T[i + 1, n]$ vsaj dolžino $PLCP[i] - 1$. Torej je $PLCP[i + 1] \geq PLCP[i] - 1$ in še vedno velja $PLCP[i] + 2i < PLCP[i + 1] + 2(i + 1) \leq PLCP[i] + 2i + 1$. Ker je $PLCP[n - 1] + 2(n - 1) < 2n$, saj $T[n - 1, n]$ ima lahko dolžino najdaljše skupne predpone 1, s poljubno pripono, 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 $n - 1$. Pri tem bitno polje H potrebuje dodatno podatkovno strukturo za $izbiro_1$ [6].

Vrednost $LCP[i]$ se lahko pridobi v $O(\log n)$ času. Vrednost je izračunana z uporabo formule $LCP[i] = izbira_1(H, SA[i]) - 2SA[i]$. Operacija potrebuje $O(\log n)$ časa, saj je potrebno izračunati vrednost $SA[i]$, ki se jo izračuna v $O(\log n)$ času, ostale operacije pa potrebujejo konstanten čas.

Prostorska zahtevnost polja LCP je predstavljena s sledečo lemo:

Lema 3.8. *Podatkovna struktura LCP potrebuje* $2n + o(n)$ *bitov za pravilno delovanje.*

Dokaz. Podatkovna struktura LCP je shranjena kot bitno polje $H[1, 2n - 1]$. Bitno polje H potrebuje dodatnih $o(n)$ bitov za dodatno podatkovno strukturo, ki omogoča

izvajanje operacije $izbira_1$ v konstantnem času. Ker je bitno polje H dolžine $2n-1$ in potrebuje dodatnih $o(n)$ bitov, potem celotna podatkovna struktura za shraniti polje LCP potrebuje $2n + o(n)$ bitov. □

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 3.9. *Podatkovna struktura kompaktno priponsko drevo nad besedilom 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 3.6 sledi, da je potrebnih $4n + o(n)$ bitov za shraniti topologijo drevesa τ. Iz Leme 3.8 pa sledi, da je potrebnih $2n + o(n)$ bitov za shraniti polje LCP.

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

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].

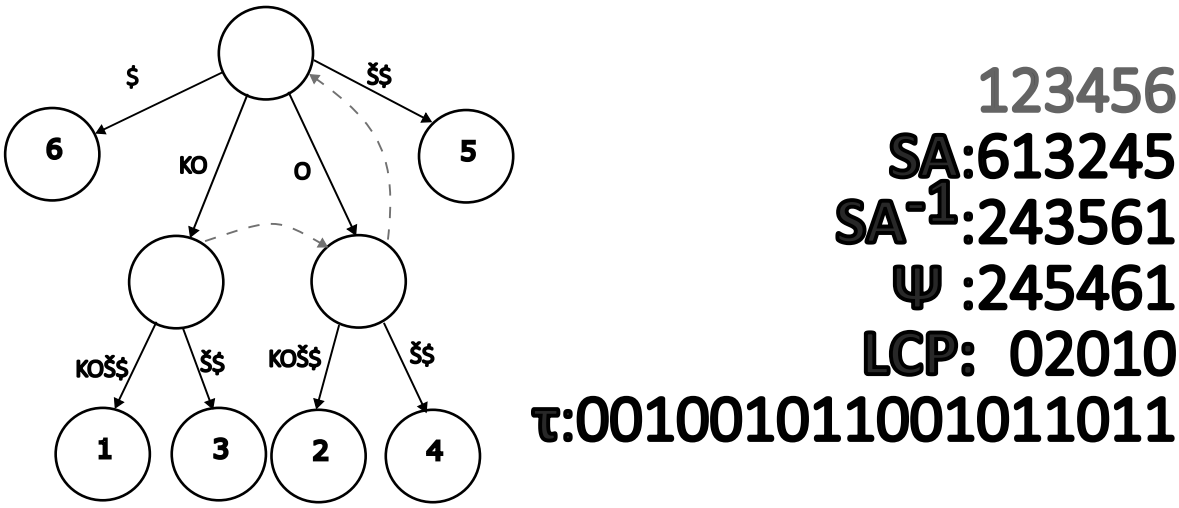
Posledica 3.10. *Kompaktno priponsko drevo implementirano s pomočjo kompaktnega priponskega polja, ki potrebuje $|CSA| = nH_h + O(n \log\log n / \log_{|\Sigma|} n)$ bitov, potrebuje $|CST| = nH_h + 6n + O(n \log\log n / \log_{|\Sigma|} n)$ bitov.*

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

Posledica 3.11. *Kompaktno priponsko drevo implementirano s pomočjo kompaktnega priponskega polja, ki potrebuje $O(\frac{1}{\epsilon} n \log|\Sigma|)$ bitov, potrebuje $|CST| = O(\frac{1}{\epsilon} n \log|\Sigma|)$ bitov. Pri tem je ϵ poljubna konstanta, ki ima vrednost $0 < \epsilon < 1$.*

Primer kompaktnega priponskega drevesa je prikazan na Sliki 10. Zaradi lažje berljivosti so vse komponente kompaktnega priponskega drevesa (na desni strani slike) prikazane v ne kompaktni obliki. Priponsko polje je prikazano v treh vrsticah (vrstica, ki se začne s SA, vrstica, ki se začne s SA^{-1}, in vrstica ki se začne s Ψ).

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 $\delta = \omega(\log_{|\Sigma|} n)$



Slika 10: Primer priponskega drevesa (levo) in kompaktnega priponskega drevesa (desno) za besedo »KOKOŠ$«.

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, ki so bile prej izvršene v konstantnem času.

3.3.1 Izgradnja

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

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

1. izgradnja kompaktnega priponskega polja SA iz besedila T,
2. izgradnja polja LCP iz kompaktnega priponskega polja SA,
3. izgradnja topologije drevesa τ iz LCP polja.

Prva zgrajena podatkovna struktura je kompaktno priponsko polje SA. Ker je priponsko polje implementirano s pomočjo funkcije Ψ (shranjena v istoimenskem polju), je potrebno naračunati polje Ψ. To je storjeno s pomočjo Burrows–Wheelerjeve preslikave (angl. *Burrows–Wheeler Transform* oziroma BWT), ki je izdelana s pomočjo

priponskega polja in besedila. Preslikavo se gradi iz leve proti desni, torej se lahko BWT izgradi postopoma z deli priponskega polja dolžine b, saj za $b = n/\log n$ je potrebnih $o(n)$ bitov dodatnega prostora in $O(n \log n)$ časa [6].

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

Ko je polje L izračunano, se kompaktno predstavitev polja 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 se ne zapiše črke $c = T[A[j] - 1]$ v polju L, ampak se jo lahko neposredno zapiše v bitno polje $B_c[kb + j] = 1$ [6].

Čas potreben za izgradnjo kompaktnega priponskega polja SA je $O(n \log n)$. Večina operacij potrebuje $O(n)$ časa. Iskanje pripon, ki spadajo v določeno vedro, potrebuje $O(n)$ časa za vsako vedro, torej potrebuje $n/b \cdot O(n) = (n \log n)/n \cdot O(n) = O(n \log n)$ časa za naračunat vse pripone.

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

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

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

Zadnja izgrajena podatkovna struktura je topologija priponskega drevesa 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.

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 besedni globini $sd(v) = LCP[i]$. Če je tako vozlišče v že v drevesu potem list i postane

njegov desni otrok. Sicer $sd(v) < LCP[i]$ in obstaja vozlišče u, ki je lahko $i-1$-vi list, za katerega velja $sd(u) > LCP[i]$. V tem primeru se ustvari novo vozlišče v', ki postane desni otrok od v ter ima dva otroka, in sicer levega otroka u ter desnega otroka listi i. Na ta način se izgradi priponsko drevo zgolj iz priponskega polja SA in LCP polja. Predstavljena metoda je implementirana z uporabo sklada, ki hrani vozlišče v ter besedno globino $sd(v)$. Na skladu se ne shrani kazalca na vozlišče, ampak se shrani predstavitev poddrevesa s 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 besedno dolžino $sd(koren) = 0$. Na sklad je dodanih največ $n-1$ vozlišč, ker ima priponsko drevo največ $n-1$ notranjih vozlišč. Da bo na koncu izgradnje sklad prazen in topologija drevesa τ pravilna, se sklad sprazni 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 $O(n \log n)$. V vsakem koraku je v topologijo drevesa dodan nov list. Za vsak list je lahko dodano novo vozlišče na sklad, če za besedno 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 3.12. *Časovna zahtevnost izgradnje kompaktnega priponskega drevesa za vhodno besedilo T je $O(n \log n)$ ter v času izgradnje je prostorska zahtevnost vedno kompaktna.*

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

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

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

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

Poglavje se lahko zaključi tako, kot se je začelo, s primerom človeškega genoma. 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 [18]. 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].

[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.

4 OPERACIJE NAD PRIPOSKIMI DREVSESI

V tem poglavju bodo analizirane implementacije operacij nad priponskimi drevesi iz Definicije 3.4, časovna zahtevnost izgradnje priponskega drevesa in prostorska zahtevnost priponskega drevesa ter kako je implementirano iskanje vzorcev s pomočjo priponskih dreves.

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

4.1 TEORETIČNA ANALIZA

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

Kompaktno priponsko drevo je lahko implementirano s pomočjo različnih različic kompaktnega priponskega polja, zato imajo operacije, ki so implementirane s pomočjo priponskega polja, različne časovne zahtevnosti. V Tabeli 4 imajo zato nekatere operacije časovno zahtevnost $O(t_\Psi)$ (časovna zahtevnost funkcije Ψ) ali $O(t_{SA})$ (časovna zahtevnost dostopa do priponskega polja) ter prostorska zahtevnost kompaktnega priponskega drevesa vsebuje člen $|CSA|$. Posledica 3.10 in Posledica 3.11 predstavljata prostorsko zahtevnost kompaktnega priponskega drevesa z uporabo dveh implementacij kompaktnega priponskega polja, zato so v Tabeli 5 predstavljene prostorske in

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

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

časovne zahtevnosti obeh implementacij priponskega polja.

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

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

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

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

Naslednja primerjava implementacij bo narejena za operacijo $jeList(v)$, ki preveri ali je vozlišče v list priponskega drevesa ali ne. V priponskem drevesu je implementirana operacija $jeList(v)$, tako da se preveri ali za vsak znak iz abecede Σ obstaja povezava do drugega vozlišča. Če ne obstaja nobena taka povezava, potem je vozlišče v list, sicer ni. V kompaktnem priponskem drevesu pa se preveri, če topologija drevesa $\tau[v, v + 1] == 01$, saj ima list v predstavitvi drevesa z uravnoteženimi oklepaji obliko 01. Obe implementacij sta prikazani v Algoritmu 3, in sicer je v zgornjem delu prikazana implementacija za priponsko drevo, v spodnjem delu pa implementacija za kompaktno priponsko drevo.

Algoritem 3: Implementacija operacije $jeList(v)$ za ST in CST

```
Vhod: Vozlišče v
(/* Implementacija za ST                                    */)
za i = 1, ..., |Σ|
    če i-ti otrok obstaja potem
        return False
return True
(/* Implementacija za CST                                   */)
če τ[v] == 0 ∧ τ[v + 1] == 1 potem
    return True
return False
```

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

$$a = besedilo(u)[globinaNiza(v) + 1] =$$
$$= S[rang_1(D, SA^{-1}[SA[rangList(u)] + globinaNiza(v)])],$$

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

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

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

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

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

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

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

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

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

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

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

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

Operacija $lca(v, w)$, ki vrne najgloblje vozlišče u, ki je hkrati predhodnik od v in od w, ima enako časovno zahtevnost v obeh primerih. Pri tem priponsko drevo potrebuje dodatno podatkovno strukturo, ki je izgrajena v času $O(n)$. V priponskem drevesu je lahko operacija $lca(v, w)$ naivno implementirana s pomočjo primerjanja vozlišč, na podoben način kot je implementirana v kompaktni predstavitvi LOUDS, kar je prikazano v Algoritmu 2. Pri tem pa se primerjata globina vozlišča in kazalca na vozlišče namesto položaj vozlišča v zaporedju. Operacija je lahko izboljšana na konstantni čas s pomočjo algoritma, ki sta ga zapisala Harel in Tarjan [19]. Dodatna podatkovna struktura L (ime je enako kot v kompaktnem priponskem drevesu) potrebuje $O(n)$ dodatnega prostora ter mora omogočati RMQ operacije. Implementirana je na podoben način kot v kompaktnem priponskem drevesu. Torej se operacija $lca(v, w)$ tako v priponskem drevesu kot tudi v kompaktnem priponskem drevesu zapiše kot

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

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

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

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

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

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

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

Algoritem 4: Iskanje intervala v SA (del CST-ja), v katerem je prisoten vzorec P, [6]

Vhod: Kompaktno priponsko drevo CST, vzorec P
Izhod: Del priponskega polja, ki se začne z P

```
1 [s, e] = [C[P[m]] + 1, C[P[m] + 1]]
2 za i = m..1
3     če s > e potem
4         return [−1, −1]
5     c = P[i]
6     [s′, e′] = [rang_1(B_c, s − 1) + 1, rang_1(B_c, e)]
7     [s, e] = [C[c] + s′, C[c] + e′]
8 return [s, e]
```

V kompaktnem priponskem drevesu se prisotnost vzorca išče s pomočjo vzvratnega iskanja (angl. *Backward Search*) vzorca v priponskem polju SA. Vzvratno iskanje za predhodno predstavljeno kompaktno priponsko polje, ki je prikazano s Algoritmom 4, potrebuje $O(mt_\Psi)$ časa, da se izvrši. Za drugačno implementacijo kompaktnega priponskega polja, se nadomesti vrstico 6 v Algoritmu 4 z binarnim iskanjem nad Ψ_c in zato je potrebno $O(m \log nt_\Psi)$ časa. Nato pa je potrebo preveriti, ali je $[s, e] \neq [-1, -1]$, za kar je potrebno konstantno časa. Torej je potrebno $O(mt_\Psi)$ časa za preveriti prisotnost vzorca ali $O(m \log nt_\Psi)$ z uporabo binarnega iskanja. V primeru, da se uporabi opisano kompaktno priponsko polje, pa se poizvedba izvrši v času $O(m)$.

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

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

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

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

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

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

V zadnjem razdelku Tabele 4 pa sta prikazani še dve primerjavi. Prva primerjava je časovna zahtevnost izgradnje priponskega drevesa ali kompaktnega priponskega drevesa iz besedila T dolžine n. Izrek 2.10 trdi, da je možno priponsko drevo izgraditi s pomočjo Ukkonenovega algoritma v času $O(n)$. Kompaktno priponsko drevo pa se po Izreku 3.12 lahko izgradi v času $O(n \log n)$. Čeprav bi se lahko kompaktno priponsko drevo izgradilo v času $O(n)$ s pomočjo priponskega drevesa. Tako izdelan algoritem ne ohranja kompaktne prostorske zahtevnosti skozi celotno izgradnjo, za razliko od predstavljenega algoritma. Algoritem za izgradnjo kompaktnega priponskega drevesa je za $O(\log n)$-krat počasnejši od algoritma za izgradnjo priponskega drevesa, vendar lahko izgradi priponsko drevo za večja vhodna besedila za razliko od Ukkonenovega algoritma za isto količino spomina.

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

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

4.2 OPIS METODE EMPIRIČNE PRIMERJAVE

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

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

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

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

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

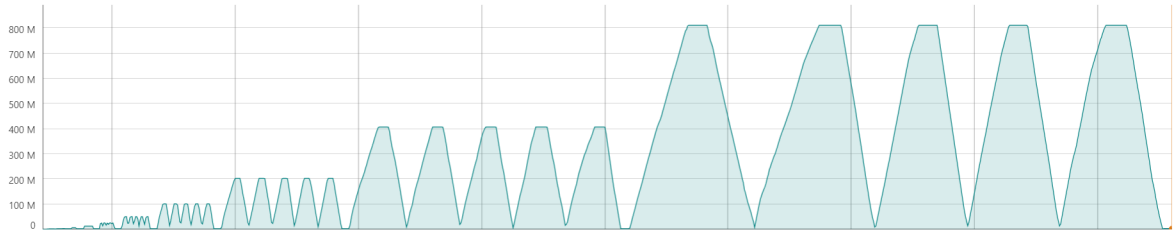
$$T_{i,v} = (t_{izg}, t_5, t_{50}, t_{500}, t_{\log n}),$$

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

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

Uporabljeni bo Bytehound [20] pomnilniški profiler, saj omogoča grafični prikaz porabe spomina v času izvajanja programa. Primer prikaza porabe delovnega spomina v času izvajanja testa je prikazan na Sliki 11. Na sliki se jasno vidi, da večanje priponskega drevesa povzroči rast zasedenega prostor na delovnem spominu v času izgradnje in poizvedbe. Na sliki se lahko opazi, da izgradnja in poizvedbe v kompaktnem priponskem drevesu potrebujejo krepko manj časa in prostora, zato izgleda, kot da se graf konča malo pred koncem vodoravne osi. Razlog za to je izgradnja in poizvedba nad kompaktnim priponskim drevesom, ki potrebuje manj pomnilnika in posledično manj časa za dodeliti in sprostiti delovni spomin.

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



Slika 11: Zasedenost spomina testiranjem različnih implementacij priponskega drevesa skozi celotno izvajanje testa.

To je tudi vidno na Sliki 11, zato ima zadnje izgrajeno prionsko drevo 5 vrhov in vsak vrh predstavlja eno ponovitev testiranja. S pomočjo profilerja je bila tudi izmerjena velikost originalnega besedila B_0, ki je prikazan v zadnjem stolpcu Tabele 6, saj se velikost razlikuje glede na vhodno besedilo.

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

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

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

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

4.2.1 Pred obdelava besedil

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

Algoritem 5: Metoda podaljševanja vhodnega besedila

Vhod: Vhodno besedilo B, želena velikost s_{max}
Izhod: Besedilo B_0

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

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

4.2.2 Iskanje vzorcev

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

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

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

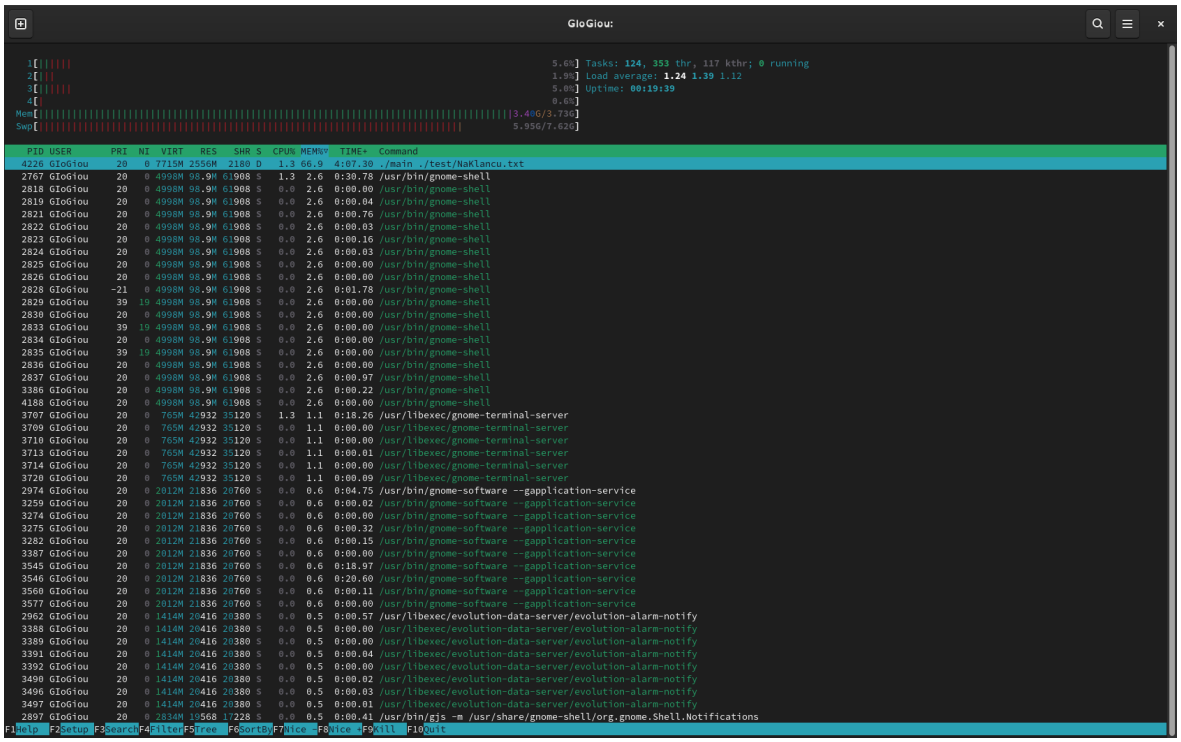
4.3 REZULTATI EMPIRIČNE PRIMERJAVE

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

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

Ob testiranju izgradnje večjih priponskih dreves je bilo opaženo, da operacijski sistem ubije (angl. *kill*) proces, ki izvaja testiranje, zaradi nevarnosti o preseganja velikosti delovnega spomina. To se je zgodilo pri izgradnji priponskega drevesa velikosti 4000000 znakov. Pri tem se je tudi opazilo, da pri priponskem drevesu velikosti 2048000 znakov ni mogoče izgraditi drevesa do konca, saj računalnik ne uspe premakniti strani iz delavnega pomnilnika na `Swap` razdelek in obratno. To naredi računalnik neodziven in proces je v neprekinjenem spanju (angl. *Uninterruptible sleep* ali stanje D), pri tem računalnik doseže 6 GB zasedenega prostora na `Swap` razdelku. Na Sliki 12 je prikazan upravljalnik opravil Htop v času izgradnje priponskega drevesa za besedilo velikosti 2048000 znakov. Proces v modri vrstici predstavlja program za testiranje izgradnje besedil. V stolpcu označenim S (Stanje ali angl. *Status*) je vidno, da je je stanje procesa označeno kot D, ker je proces v neprekinjenem spanju. Po več kot 5 minutah od začetka izgradnje prvega priponskega drevesa sem se odločil, da se proces ubije. Posledično se je znižala velikost besedil za izgradnjo zadnjega priponskega drevesa na 1024000 znakov.

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



Slika 12: Posnetek zaslona upravljalnika opravil Htop med izgradnjo priponskega drevesa za besedilo dolžine 2048000 znakov.

Pri tem se je tudi omejila velikost besedila B_0 na $S_{max} = 25000000$ znakov. To zagotavlja dovolj znakov za izgradnjo vhodnih besedil ter za izgradnjo besedil, ki bodo predstavljali vzorce. Ker je DNK sekvenca [9] daljša od velikosti S_{max}, je bila le ta odrezana na velikost S_{max} in sicer bo uporabljeni le prvih 25000000 znakov. Za razliko od DNK sekvence, je besedilo Na klancu [10] moralo biti podaljšano na $S_{max} = 25000000$ znakov. To je bilo storjeno z implementacijo Algoritma 5 v programskem jeziku C++. Pri tem ni bilo potrebno implementirati preverjanja, ali šestkratnik števca i presega velikost besedila B. Torej tega nisem storil.

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

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

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

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

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

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

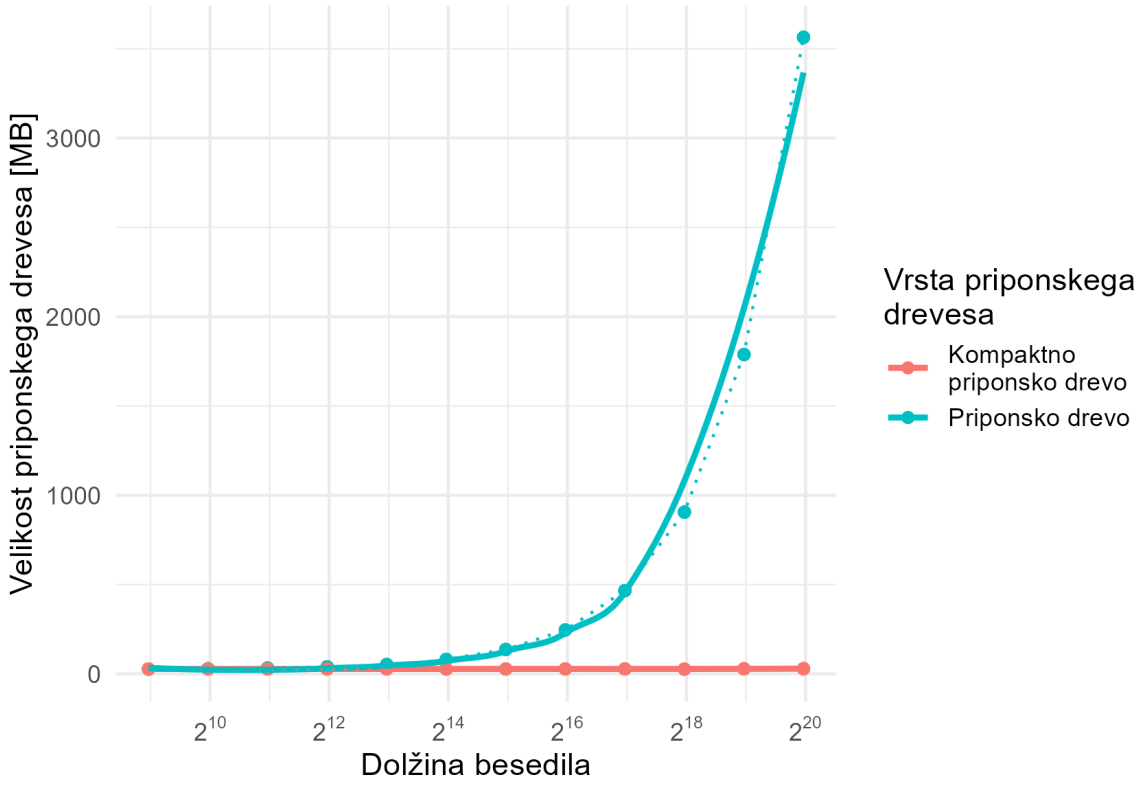
Ker tako prevedena datoteka ne omogoča neposrednega nadzorovanja velikosti podatkovnih struktur med izvajanjem, bo za ta namen uporabljen profiler spomina. Zato je potrebo pred začetkom testiranja priponskih dreves zagnati tudi profiler, kar je storjeno z ukazom:

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

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

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

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

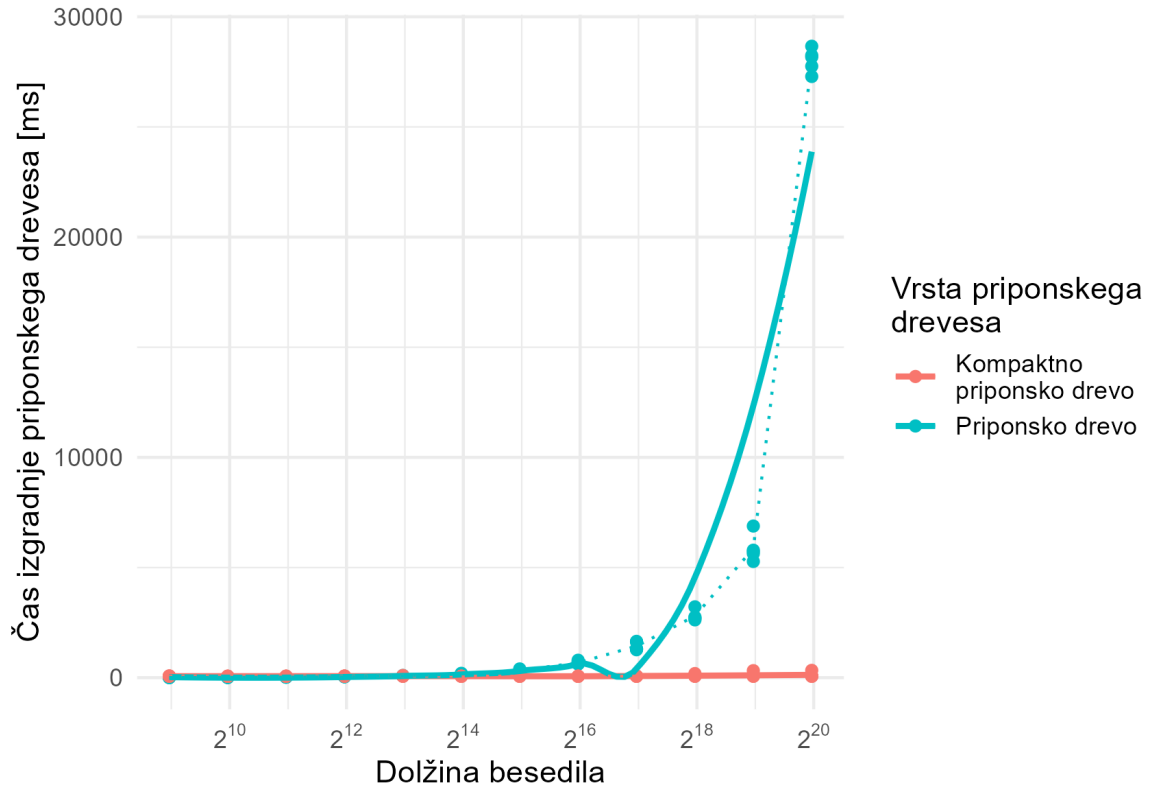


Slika 13: Graf velikosti priponskega drevesa izgrajenih iz besedil različne velikosti. Vhodno besedilo je DNK sekvenca.

Na Sliki 13 je prikazana potrebna količina delovnega spomina za izgradnjo priponskega drevesa, ki je označena z modro, ter kompaktnega priponskega drevesa, ki pa je označena z rdečo barvo. Na sliki je uporabljena, na vodoravni osi, logaritmična skala za velikost besedila, saj je vsako naslednje testirano besedilo dvakrat večje od predhodnega. Iz slike se lahko jasno vidi, da za besedila do dolžine 8000 znakov obe drevesi zasedeta približno enako količino prostora na delovnem spominu, in sicer priponsko drevo zasede $39,8$ MB in kompaktno priponsko drevo zasede $27,4$ MB. Vse nadaljnje razlike v velikosti med priponskim drevesom in kompaktnim priponskim drevesom so

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

Na Sliki 13 se lahko tudi opazi, da velikost priponskega drevesa ne presega 4 GB, kar je velikost notranjega spomina. Pri tem pa je potrebno upoštevati dejstvo, da operacijski sistem potrebuje 1, 76 GB delovnega spomina takoj po zagonu računalnika. To pomeni, da čeprav velikost priponskega drevesa ne presega velikosti delovnega pomnilnika, mora biti le ta shranjen v `Swap` razdelku, saj mora računalnik v vsakem trenutku zagotoviti dovolj prostora na delovnem pomnilniku za operacijski sistem, torej ima program na razpolago največ 2, 24 GB delovnega pomnilnika. Zadnje testirano priponsko drevo potrebuje 3, 56 GB dolgovnega pomnilnika, zato mora biti vsaj 1, 22 GB drevesa shranjenega v na `Swap` razdelku.



Slika 14: Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je DNK sekvenca.

Ker je zadnje izgrajeno priponsko drevo, moralo biti shranjeno tudi na `Swap` razdelku, je potrebno preveriti, na kakšen način to vpliva na čas izgradnje drevesa ter iskanje v njem. Rezultati testiranja časa izgradnje različnih implementacij priponskega drevesa so prikazani na Sliki 14, kjer z modro barvo je prikazan čas potreben za izgradnjo priponskega drevesa ter z rdečo pa čas potreben za izgradnjo kompaktnega priponskega drevesa. Vsi časi, ki so prikazani na sliki, so v milisekundah.

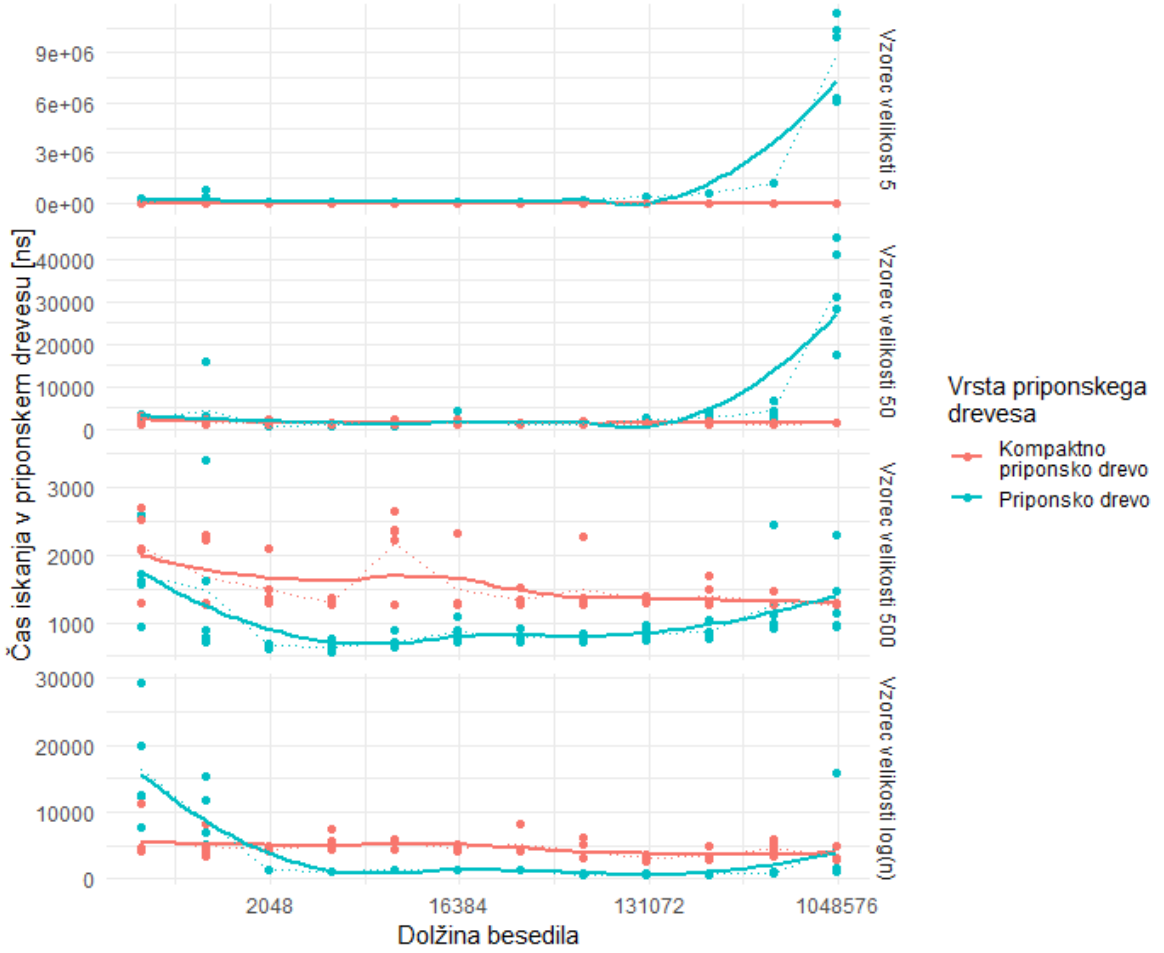
Pri izgradnji največjega priponskega drevesa se lahko opazi na Sliki 14, da je potrebni čas višji od pričakovanega. Skok v časovni zahtevnosti se pojavi, pri izgradnji priponskega drevesa, ki je moral biti delno shranjen na `Swap` razdelku. Vsa ostala priponska drevesa so zgrajena dvakrat počasneje kot predhoden priponsko drevo, kar je tudi pričakovano, saj Ukkonenov algoritem izgradi priponsko drevo v času $O(n)$. Torej, ker se je velikost besedila podvojila, potemtakem se bo tudi čas izgradnje podvojil. Izjema je zadnje izgrajeno priponsko drevo (dolžina vhodnega besedila je 1024000 znakov), kjer se čas izgradnje poveča za $4,78$-krat. Iz tega se lahko sklepa, da uporaba `Swap` razdelka negativno vpliva na čas izgradnje priponskega drevesa.

Iz Slike 14 je tudi razvidno, da čas potreben za izgradnjo kompaktnega priponskega drevesa raste počasneje kot čas izgradnje priponskega drevesa. Čas potreben za izgradnjo kompaktnega priponskega drevesa za besedilo dolžine 500 znakov je $70,8$ milisekund, čas za izgradnjo kompaktnega priponskega drevesa za besedilo dolžine 1024000 znakov pa je 121 milisekund. Kompaktno priponsko drevo se zgradi hitreje od priponskega drevesa pri besedilih, ki so daljši od 8000 znakov. Priponsko drevo potrebuje $83,2$ milisekunde za izgradnjo, za razliko od kompaktnega priponskega drevesa, ki pa potrebuje $75,2$ milisekunde za biti izgrajeno nad besedilom dolžine 8000 znakov.

Nazadnje se lahko še analizira vpliv velikosti priponskega drevesa na potreben čas za izvršiti poizvedb nad njim. Na Sliki 15 so prikazani časi, v nanosekundah, potrebni za iskanje vzorcev, in sicer z modro barvo so predstavljeni časi za poizvedbe v priponskih drevesih, z rdečo pa so predstavljeni časi za poizvedbe v kompaktnem priponskem drevesu. Na sliki so predstavljeni rezultati za vse štiri velikosti vzorcev iskanih v priponskem drevesu: 5 znakov, 50 znakov, 500 znakov in $\log n$ znakov, pri čemer je n dolžina besedila.

Na Sliki 15 so prikazani rezultati primerjave iskanja vzorcev dolžine 5 znakov. Časi potrebni za iskanje vzorca so za obe implementaciji priponskih dreves podobni in konstantni. Edina vidna razlika je v času, ki je potrebnem za najdi vzorec v priponskem drevesu dolžine 1024000 znakov. To je posledica uporabe `Swap` razdelka, zato se potrebuje $7,46$-krat več čas od ostalih poizvedb s priponskim drevesom. Ostale poizvedbe, za vzorce dolžine 5 znakov, potrebujejo približno 1000 mikrosekund, pri čemer pa priponsko drevo za besedilo dolžine 1024000 znakov pa potrebuje $8,79$ milisekund za preveriti prisotnost vzorca. Čas, ki je potreben za poizvedbo v kompaktnih priponskih drevesih, je konstanten in je približno $3,2$ mikrosekunde.

Za vzorce dolžine 50 znakov je iz Slike 15 razvidno, da so potrebni časi za iskanje podobni kot pri iskanju vzorcev dolžine 5 znakov. Torej je čas, ki je potreben za iskanje v priponskem drevesu, konstanten in je približno $2,2$ mikrosekunde. Ko priponsko drevo preraste delovni spomin in mora bit delno shranjen na `Swap` razdelku se iskalni čas poveča za $7,44$-krat. Pri tem iskanje vzorcev dolžine 50 znakov potrebuje približno enako časa kot iskanje vzorcev v kompaktnem priponskem drevesu. V kompaktnem



Slika 15: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je DNK sekvenca.

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

Iskanje prisotnosti vzorca dolžine 500 znakov, rezultati so prikazani na Sliki 15 kot predzadnji graf, je bolj učinkovito s priponskimi drevesi. Čeprav je za vse testirane velikosti priponskega drevesa iskanje v priponskem drevesu hitrejše ali primerljivo z iskanjem v kompaktnem priponskem drevesu, se lahko opazi rast v potrebnem času v večjih priponskih drevesih, ki so deloma shranjena v `Swap` razdelku. Čeprav razlika ni tako očitna kot v primeru iskanja vzorca dolžine 5 ali 50 znakov, se lahko vseeno vidi rast iz grafa. Možen razlog za nižjo rast je prisotnost strani (angl. *page*), ki vsebujejo vzorec v delovnem spominu in ne na `Swap` razdelku. Pri tem je vseeno potrebna 1 mikrosekunda za najti vzorec dolžine 500 znakov. Čas iskanja vzorcev v kompaktnih priponskih drevesih še vedno ostane konstantno, pri čemer se potreben čas za iskanje

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

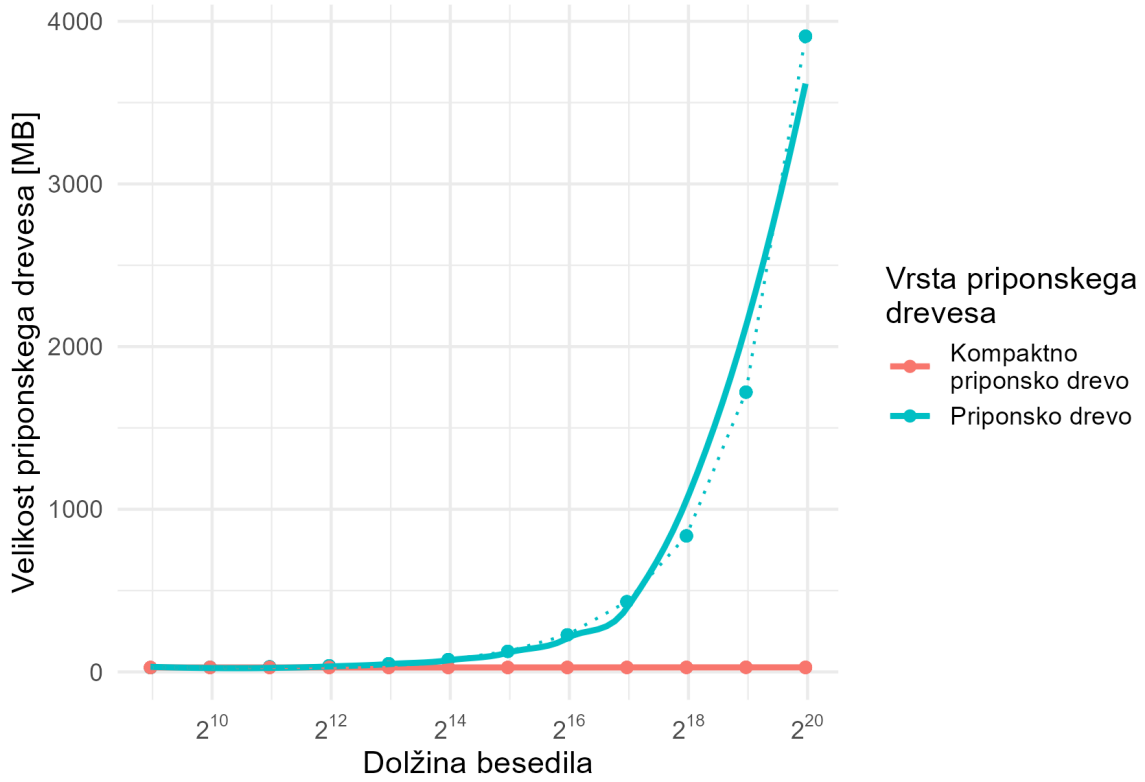
Zadnje testiranje, ki je prikazano na Sliki 15, je iskanje vzorcev dolžine $O(\log(n)$, pri čemer je n velikost besedila, ki je predstavljeno s priponskim drevesom. Velikosti iskanih vzorcev se gibajo od 9 znakov do 20 znakov. Iz grafa je razvidno, da kompaktno priponsko drevo, označeno na grafu z rdečo barvo, potrebuje približno konstanto časa za najti vzorec v priponskem drevesu. Za iskanje vzorca potrebuje približno $4,5$ mikrosekunde. Pri tem pa je zgodba bistveno drugačna pri iskanju vzorcev s pomočjo priponskega drevesa, ki je prikazano z modro barvo na sliki. Za vzorce velikosti 9 in 10 znakov je čas iskanja v priponskem drevesu višji kot čas iskanja v kompaktnem priponskem drevesu. V vseh ostalih primerih pa je čas iskanja nižji kot v kompaktnem priponskem drevesu, podobno kot pri iskanju vzorcev dolžine 500 znakov. Iskanje vzorca dolžine 11 znakov ali več potrebuje približno 1 mikrosekundo, da se izvrši. Pri tem se lahko tudi opazi, da je čas iskanja v zadnjem drevesu $4,5$-krat višji zaradi uporabe `Swap` razdelka, kar pomeni, da potrebuje približno enako časa kot iskanje v ekvivalentnem kompaktnem priponskem drevesu. Iz tega testa se lahko sklepa, da za iskanje krajših vzorcev (vzorci do dolžine 10 znakov) je boljše uporabiti kompaktno priponsko drevo, sicer pa je boljše uporabiti priponsko drevo, če velikost delovnega spomina to omogoča ter če je število iskanih vzorcev dovolj veliko. S tem se lahko amortizira čas iskanja vzorcev, pri čemer je število vzorcev $O(n)$, kar zniža čas izgradnje drevesa za vsak vzorec na $O(1)$.

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

Druga primerjava je bila narejena s Cankarjevim romanom Na klancu [10]. Namen te primerjave je primerjati lastnosti priponskega drevesa nad večjo začetno abecedo, kot je na primer abeceda naravnega jezika. Ker slovenščina uporablja ne ASCII znake, je potrebno pred samim začetkom testiranja besedilo pred pripraviti. To je bilo storjeno tako, da so bili vsi ne ASCII znaki odstranjeni oziroma zamenjani z ASCII alternativami. Na primer znak, ki predstavlja '...', je bil zamenjan s tremi pikami, vse črke z naglasi (kot so strešice, ostrivci ter drugi) so bile zamenjani z osnovnim znakom, torej

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

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

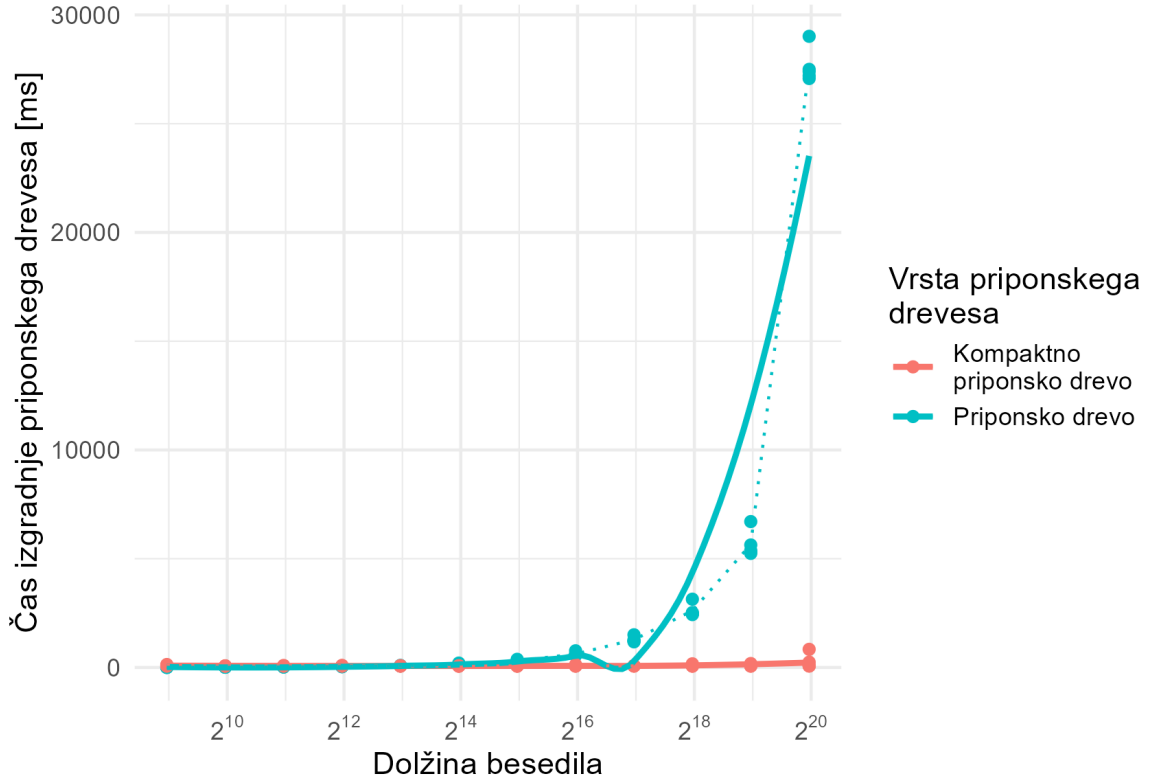


Slika 16: Graf velikosti priponskega drevesa izgrajenih iz besedil različnih velikosti. Vhodno besedilo je roman Na klancu.

Podobno kot za DNK sekvenco je bila izmerjena velikost, ki je potrebna za priponsko drevo na delavnem spominu. Rezultati meritve so prikazani na Sliki 16, kjer je z modro barvo prikazana velikost priponskega drevesa z rdečo barvo pa velikost kompaktnega priponskega drevesa. Za to besedilo je velikost prionskega drevesa manjša ali približno enaka (manj kot $1,5$-krat večja od kompaktnega prionskega drevesa) kompaktnemu priponskemu drevesu, do besedila dolžine 4000 znakov. Velikosti priponskih dreves rastejo dokler ne dosežejo velikosti $3,9$ GB za besedilo dolžine 1024000 znakov. Čeprav

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

Iz Slike 16 je razvidno, da je priponsko drevo prostorsko bolj učinkovito za besedila, ki so krajša od 4000 znakov. Za vsa ostala daljša besedila je bolj prostorsko učinkovito uporabiti kompaktno priponsko drevo. To pa je zgolj eden od pogojev, kako izbrati primerno podatkovno strukturo.

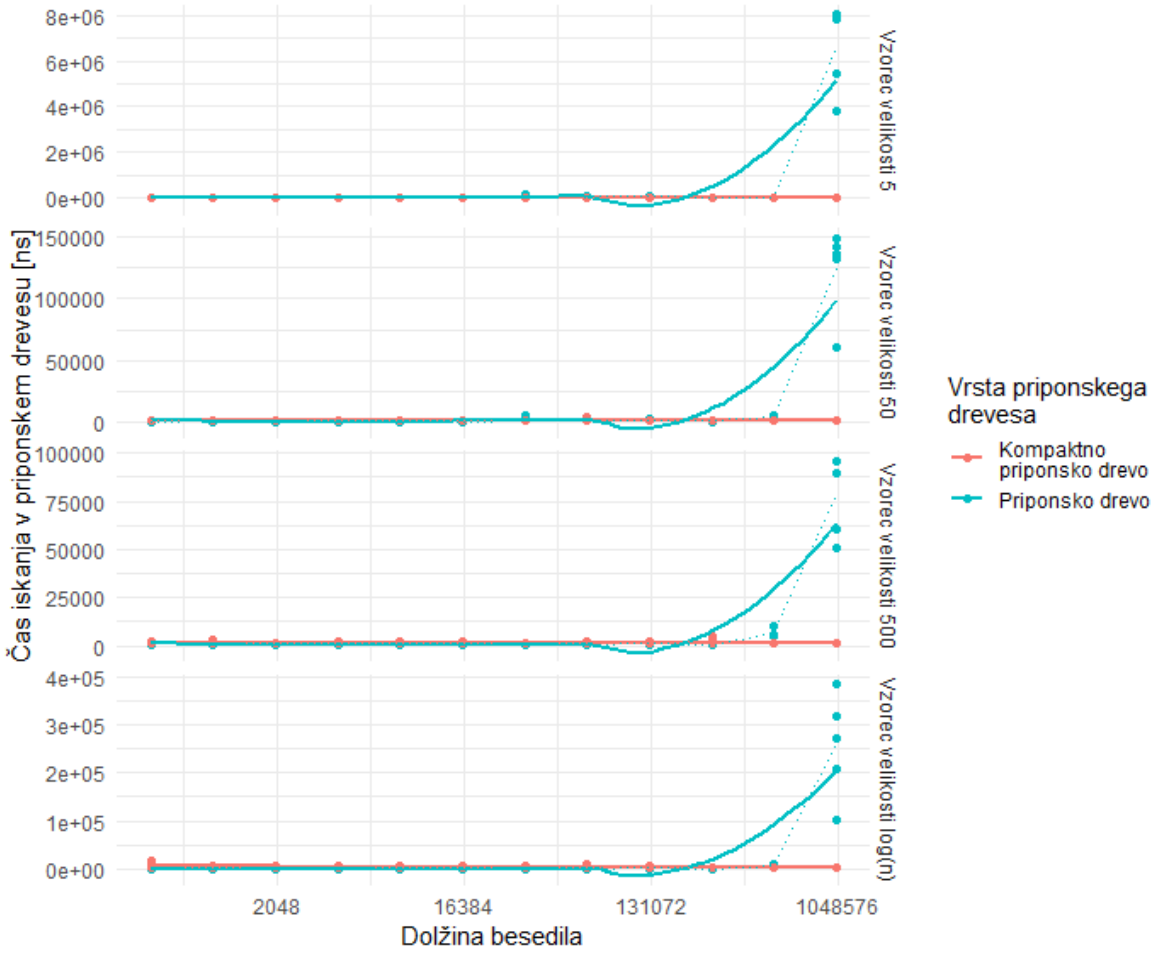


Slika 17: Graf prikazuje čas izgradnje priponskega drevesa za različne dolžine vhodnih besedil. Vhodno besedilo je roman Na klancu.

Naslednji pogoj, ki ga je potrebno upoštevati, je čas izgradnje priponskega drevesa, saj za je nekatere operacije bolj pomembno hitro izgraditi priponsko drevo, kot zasedati manj prostora na delovnem pomnilniku s priponskim drevesom. Na Sliki 17 je prikazan potreben čas za izgradnjo priponskega drevesa, ki je označen z modro barvo, ter čas potreben za izgradnjo kompaktnega priponskega drevesa, ki pa je označeno z rdečo barvo. Iz grafa je razvidno, da čas, ki je potreben za izgradnjo priponskega drevesa, je v vsakem koraku podvojen. Pri tem pa se razlikuje priponsko drevo nad besedilom velikosti 1024000 znakov, ki potrebuje $4,88$-krat več časa za izgradnjo, kot priponsko

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

Torej iz Slike 17 se lahko opazi, da je za besedila krajša od 8000 znakov čas izgradnje priponskega drevesa krajši kot za kompaktno priponsko drevo. Za daljša vhodna besedila pa se časovno izplača uporabiti kompaktno priponsko drevo.



Slika 18: Graf prikazuje čas iskanja vzorcev različnih dolžin v različnih implementacijah priponskega drevesa. Vhodno besedilo je roman Na klancu.

Čas izgradnje priponskega drevesa vpliva na hitrost iskanja vzorcev v besedilu tedaj, ko se ne išče dovolj vzorcev v besedilu, v tem primeru se čas izgradnje ne amortizira na konstanten dodaten čas iskanja posamičnega vzorca. V tem primeru je bolj pomemben čas iskanja vzorcev v besedilu. Zato se nazadnje lahko pogleda vpliv velikosti priponskega drevesa na potreben čas za izvršiti poizvedbo nad njimi. Na Sliki 18 so prikazani časi v nanosekundah potrebni za iskanje vzorcev, in sicer z modro barvo so predstavljeni časi za poizvedbe v priponskih drevesih, z rdečo barvo pa so predstavljeni časi

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

Na zgornjem grafu Slike 18 so prikazani rezultati iskanja vzorca dolžine 5 znakov. V priponskih drevesih je časovna zahtevnost iskanja vzorca konstantna skozi celotno izvajanje. Pri tem pa so bili izmerjeni 3 različni časi. Prvi izmerjen čas je približno $2,6$ mikrosekund, ki je potreben za preveriti prisotnost vzorca v besedilih do 32000 znakov. V besedilih med 64000 in 512000 znakov je čas iskanja $15,5$-krat večji, in sicer približno 45 mikrosekund. Najbolj verjetni razlog za ta skok v potrebnem času je dolžina niza, ki ga predstavljajo povezave med vozlišči. Nizi postanejo vedno krajši in zato je potrebno več časa za binarna iskanja v naslednje povezave. Drugi razlog za daljše iskanje je večja verjetnost prisotnosti vzorca v besedilu, saj imajo krajši vzorci v naravnem jeziku večjo verjetnost, da se ponovijo, ter vzorci, ki so narejeni iz podaljšanega besedila, so zagotovo prisotni v originalnem besedilu, razen če je vzorec lih zlepek dveh podnizov originalnega besedila. Zadnji izmerjen čas, pa je čas, ki je potreben za iskanje vzorca v besedilu dolžine 1024000 znakov. Le ta čas je $260,7$-krat večji od predhodnega potrebnega časa, kar pomeni, da za najti vzorec je potrebno $6,59$ milisekunde (6588 mikrosekunde). Razlogi za to povečavo v času so enaki kot za prejšnjo povečavo v času ter uporaba `Swap` razdelka, za shranjevanje priponskega drevesa.

Za razliko od priponskega drevesa, iskanje vzorca dolžine 5 v kompaktnem priponskem drevesu potrebuje konstantni čas, da se izvrši skozi celotno testiranje, kot je to lahko razvidno iz Slike 18. Iskanje v kompaktnem priponskem drevesu potrebuje približno enako časa kot iskanje v priponskih drevesih nad besedilo krajšim od 32000 znakov, za kar je potrebno približno 3 mikrosekunde. Iz tega sledi, da za besedila krajša 8000 znakov je iskanje v priponskem drevesu boljše, saj porabi manj prostora in manj časa, da se izgradi. Iskanje v besedilih do 32000 znakov je ekvivalentno natanko tedaj, ko je možno amortizirati čas izgradnje priponskega drevesa s količino iskanih vzorcev. V daljših primerih pa je boljše uporabiti kompaktno priponsko drevo za iskanje krajših vzorcev.

Naslednja testirana poizvedba je iskanje vzorcev dolžine 50 znakov, kar je možno videti na drugem grafu Slike 18. Opazi se, da je čas, ki je potreben za izvrši poizvedbo, konstanten skozi celotno izvajanje bodisi za kompaktno priponsko drevo, bodisi za priponsko drevo. Iskanje vzorcev dolžine 50 znakov s pomočjo priponskega drevesa potrebuje približno $1,5$ mikrosekund. Z uporabo kompaktnega priponskega drevesa pa je potrebnih približno $1,2$ mikrosekunde. Pri tem je samo ena izjema, in sicer priponsko drevo, ki mora biti delno shranjeno na `Swap` razdelku. Uporaba `Swap` razdelka na trdem disku poveča čas iskanja vzorca za $27,3$-krat, kar pomeni, da je potrebno 124 mikrosekund za preveriti, ali je vzorec prisoten v besedilu ali ni. Iz rezultatov

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

Rezultati testiranja iskanja vzorca dolžine 500 znakov v besedilu, ki uporablja naravni jezik, so prikazani na tretjem grafu Slike 18. Podobno kot v predhodnih testiranjih je čas iskanja v obeh primerih konstanten. Priponsko drevo potrebuje približno 1 mikrosekundo za preveriti obstoj vzorca, kompaktno priponsko drevo pa potrebuje $1,5$ mikrosekund. Pri tem pa obstaja ena izjema, in sicer priponsko drevo, ki je delno shranjeno na `Swap` razdelku. To priponsko drevo potrebuje $78,5$ mikrosekund za izvesti poizvedbo, kar je $12,4$-krat več čas kot v predhodnem priponskem drevesu ali pa $52,3$-krat več čas kot povprečni čas ostalih poizvedb. Iz rezultatov tega testiranja je zaključek isti kot v predhodnem testiranju (iskanje vzorcev dolžine 50). To pomeni, da če se lahko priponsko drevo shrani v celoti na delovnem spominu in število iskanih vzorec je dovolj veliko, da se lahko z vsako poizvedbo amortizira čas iskanja vzorcev, se lahko uporablja priponsko drevo, sicer je bolj priporočljivo uporabiti kompaktno priponsko drevo.

Zadnje izdelano testiranje je iskanje vzorcev dolžine $O(\log n)$ v besedilu. Rezultati tega testiranja so predstavljeni na spodnjem grafu Slike 18. Podobno kot v primeru DNK sekvence so vzorci dolgi od 9 do 20 znakov. Skozi celotno izvajanje testiranja je čas poizvedbe v obeh primerih konstanten. Pri tem potrebuje poizvedba v priponskem drevesu približno $3,5$ mikrosekund, v kompaktnem priponskem drevesu pa potrebuje ista poizvedba približno $4,5$ mikrosekund. Izjema je priponsko drevo delno shranjeno v `Swap` razdelku, ki potrebuje $38,1$-krat več časa od predhodnega priponskega drevesa ali $70,4$-krat več časa od povprečnega časa poizvedbe. Torej potrebuje 257 mikrosekund za izvršiti poizvedbo. Podobno kot pri ostalih testiranjih nad besedilom v naravnem jeziku se lahko tudi iz teh rezultatov sklepa, da je priporočljivo uporabljati priponsko drevo za iskanje vzorcev v besedilu, če se lahko čas izgradnje le tega amortizira s količino iskanih vzorcev ter se lahko shrani celotno priponsko drevo v delovni spomin. Sicer pa je bolje, da se uporabi kompaktno priponsko drevo.

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

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

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

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

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

5 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.

6 LITERATURA IN VIRI

[1] E. Ukkonen, On-line construction of suffix trees. *Algorithmica* 14 (1995) 249–260. *(Citirano na straneh 7, 12 in 14.)*

[2] K. Sadakane, Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems* 41 (2007) 589–607. *(Citirano na straneh 26, 27, 28 in 31.)*

[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 44, 50, 51 in 61.)*

[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 7, 8 in 14.)*

[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 44.)*

[6] G. Navarro, *Compact Data Structures: A Practical Approach*, Cambridge University Press, 2016. *(Citirano na straneh 1, 17, 18, 20, 21, 22, 23, 28, 29, 30, 33, 34, 42 in 44.)*

[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 3.)*

[8] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997. *(Citirano na strani 3.)*

[9] *Pizza&Chili Corpus – Compressed Indexes and their Testbeds*, `https://pizzachili.dcc.uchile.cl/`. (Datum ogleda: 8. 10. 2024.) *(Citirano na straneh 46, 49 in 51.)*

[10] I. Cankar, *Na Klancu*, Genija, 2012. *(Citirano na straneh 46, 49 in 55.)*

[11] S. Gog, T. Beller, A. Moffat in M. Petri, From Theory to Practice: Plug and Play with Succinct Data Structures. V *13th International Symposium on*

Experimental Algorithms, (SEA 2014), 2014, 326–337. *(Citirano na straneh 48 in 50.)*

[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 48.)*

[13] E. Miller, *Genome — Knowledge Hub*, `https://www.genomicseducation.hee.nhs.uk/genotes/knowledge-hub/genome`. (Datum ogleda: 30. 10. 2024.) *(Citirano na strani 16.)*

[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 strani 27.)*

[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 28.)*

[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 28, 31, 37 in 44.)*

[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 28, 31 in 37.)*

[18] L. M S. Russo, G. Navarro in A. L. Oliveira, Fully-Compressed Suffix Trees. *LATIN 2008: Theoretical Informatics* (2008) 362–373. *(Citirano na straneh 31 in 35.)*

[19] D. Harel in R. E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing* 13 (1984) 338-355. *(Citirano na strani 41.)*

[20] Koute, *koute/bytehound: A memory profiler for Linux.*, `https://github.com/koute/bytehound`. (Datum ogleda: 30. 10. 2024.) *(Citirano na strani 45.)*

[21] A. Jacobs, The Pathologies of Big Data: Scale up your datasets enough and all your apps will come undone. What are the typical problems and where do the bottlenecks generally surface?. *Queue* 7 (2009) 10–19. *(Citirano na strani 61.)*