Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Številska drevesa

osnove, PATRICIA, LC Trie

Osnove

- rekurzivna podatkovna struktura
- ključi so tako veliki, da ne moremo na enkrat primerjati dveh ključev (npr. nizi črk ali nizi bitov)
- zato organiziramo podatkovno strukturo tako, da rekurzivnost ni definirana na podlagi celotnih ključev, ampak na osnovi ene črke ključa
- primerjave nas vodijo od korena do lista, le da so sedaj primerjave narejene po bitih
- elemente *shranimo v liste*, medtem ko notranja vozlišča uporabimo samo za usmerjanje poti (za razdelitev na podmnožice)
- velikost črke ključa je poljubna: črka abecede (A..Ž), nukleotid v DNK (A, C, G, T), en bit (bitno/binarno primerjanje) – v splošnem imamo abecedo Σ
- takšni strukturi rečemo *trie*, ker je uporabna za iskanje `retrieval` (E. Fredkin)
- šteli bomo število poizvedovanj po črki ključa (dostopov) in ne primerjav

Primer

- imamo binarne (bitne) črke abecede $\{0, 1\}$ in imejmo bitno predstavitev naslednjih ključev:

črka	*bitna predstavitev*	*črka*	*bitna predstavitev*
A	00001	C	00011
E	00101	G	00111
H	01000	I	01001
J	01010	K	01011
L	01100	M	01101
N	01110	O	01111
P	10000	R	10010
S	10011	X	11000
Z	11010		

- Imejmo ključe

$$\{\mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{L}, \mathtt{M}, \mathtt{R}, \mathtt{S}\}$$

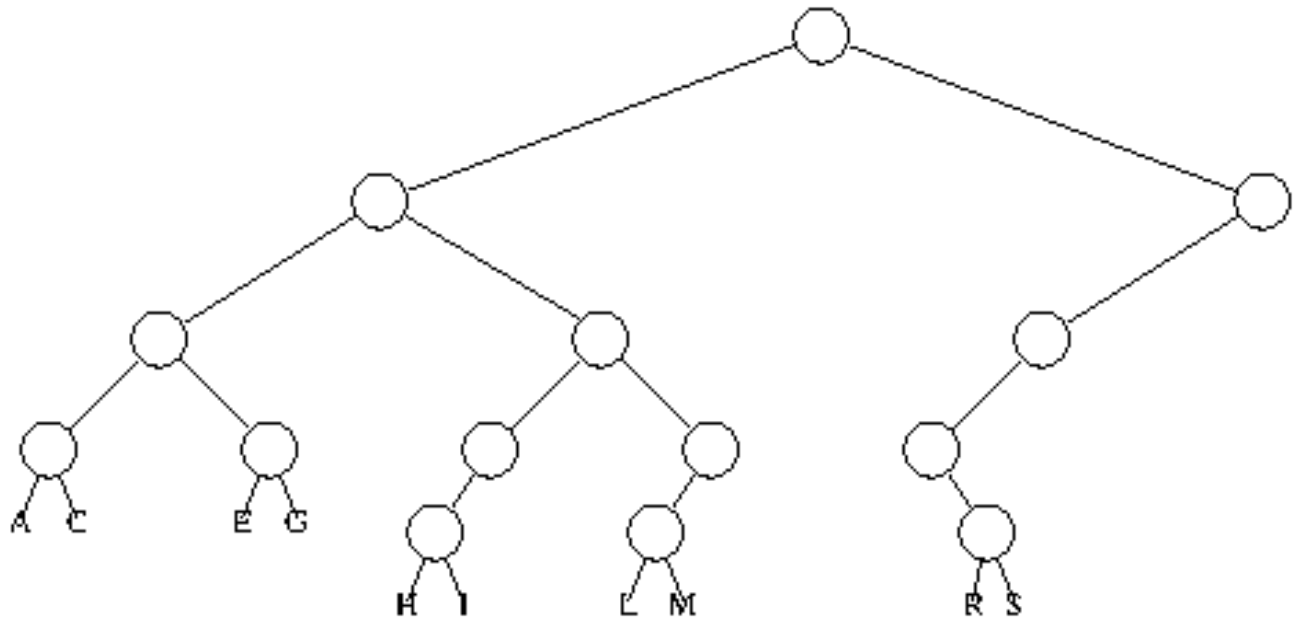
Gradnja

- črke jemljemo po vrsti od prve naprej
- ob vsaki črki razdelimo množico na dva ($|\Sigma|$) podmnožic, dokler ni v podmnožici samo še en element

Definirali smo invarianco podatkovne strukture.

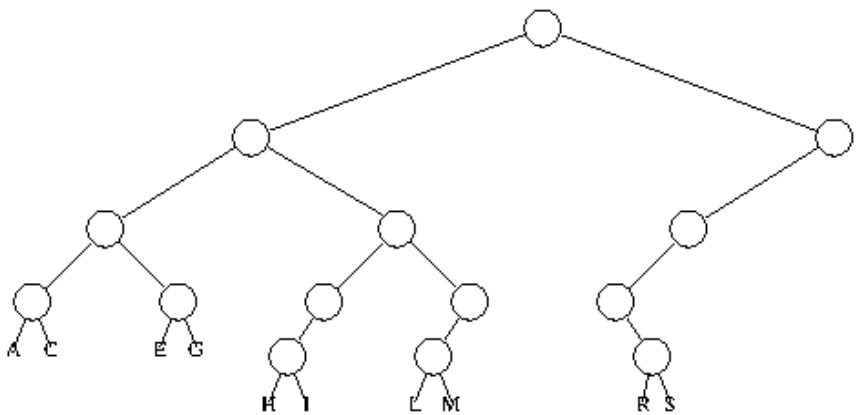
Imamo ključe:

$$\{\mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{L}, \mathtt{M}, \mathtt{R}, \mathtt{S}\}$$



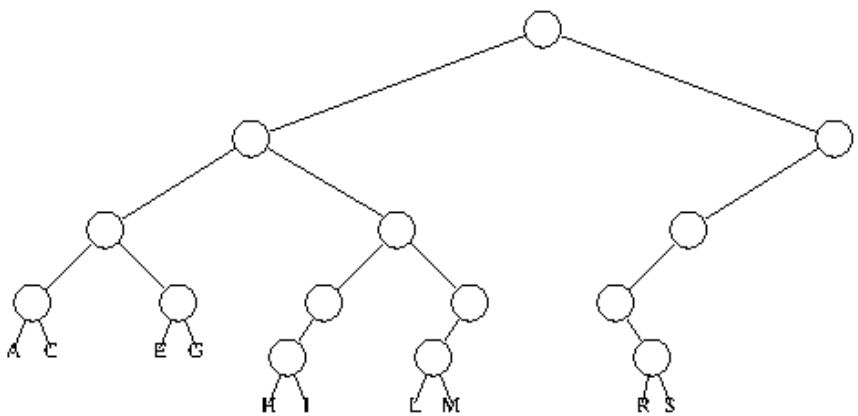
Iskanje

- iščimo L = 01100:
  - pogledamo prvi bit in je 0, zato gremo v levo podstrukturo (pod-trie)
  - pri naslednjih dveh bitih gremo obakrat desno, kjer je njuna vrednost 1
  - na koncu še dvakrat levo in smo našli L
- iščimo J = 01010:
  - pogledamo prve tri bite in gremo levo, desno in levo
  - pri četrtem bitu ne moremo desno, ker tam ni poddrevesa, torej J ni v strukturi



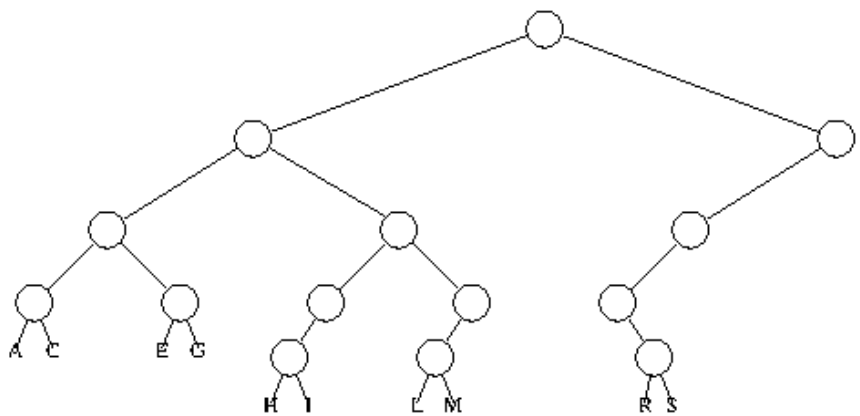
Iskanje malce drugače

- če nadaljujemo z iskanjem po obstoječi poti levo, pridemo do ključa `I`
- ključ `I` je *sosed* ključa `J`; celo *najbljižji sosed*
- toda, če iščemo ključ `K` = `01011` tudi najdemo ključ `I`, ki pa ni več najbližji sosed



Iskanje soseda

- takšnemu iskanju rečemo tudi iskanje najboljšega ujemanja ali najboljšega odgovora
- v obeh primerih smo našli *levega soseda*
- kako v splošnem najdemo levega soseda
- poiščimo še: P $= 10000$, O $= 01111$ in I $= 01001$



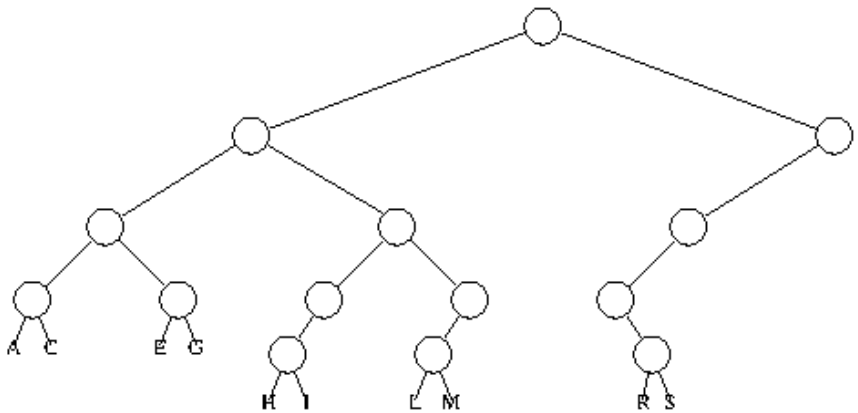
Iskanje levega, desnega in najbližjega soseda

- iskanje desnega soseda, ali najmanjšega elementa v strukturi, ki je že večji od iskanega elementa:

  Zadnjič, ko sem šel v strukturi levo, pojdi desno in potem kar se dâ levo.

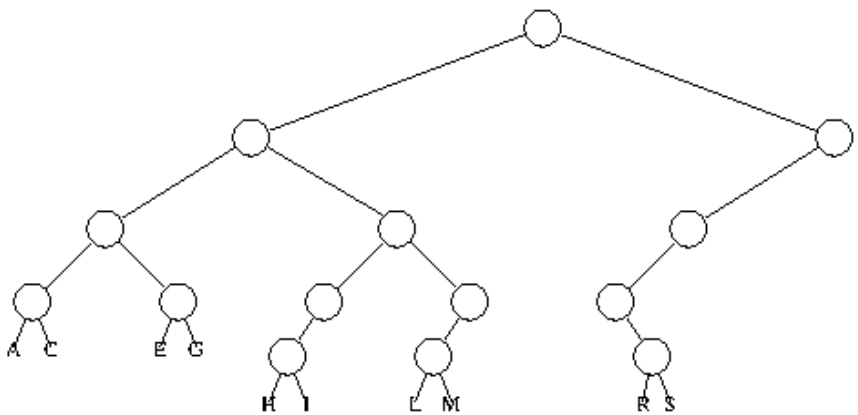
  Je to vedno prav? Robni primeri? ...

- in iskanje levega soseda – največjega elementa v strukturi, ki je še manjši od iskanega elementa
- kaj pa iskanje najbližjega soseda?



Operacije

- običajne operacije: `Najdi`, `Dodaj` in `Izloci`
- dodatne operacije: `NajdiManj`, `NajdiVec` in
- posplošena operacija: `Najdi`, ki sedaj najde najbolj podobni element v strukturi

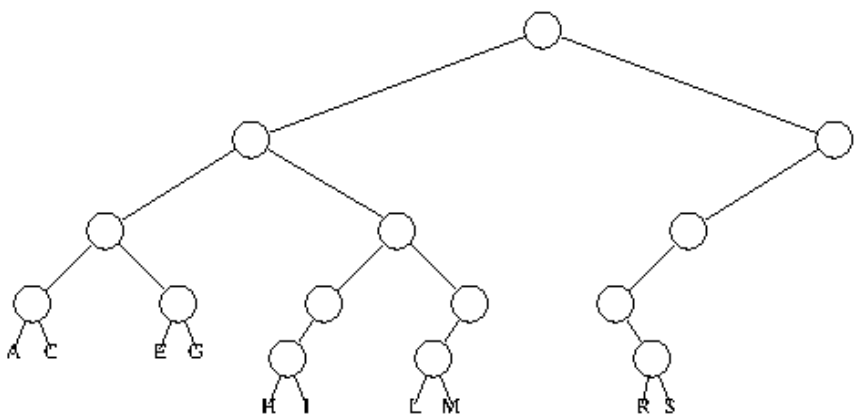


Vstavljanje

Invarianca:

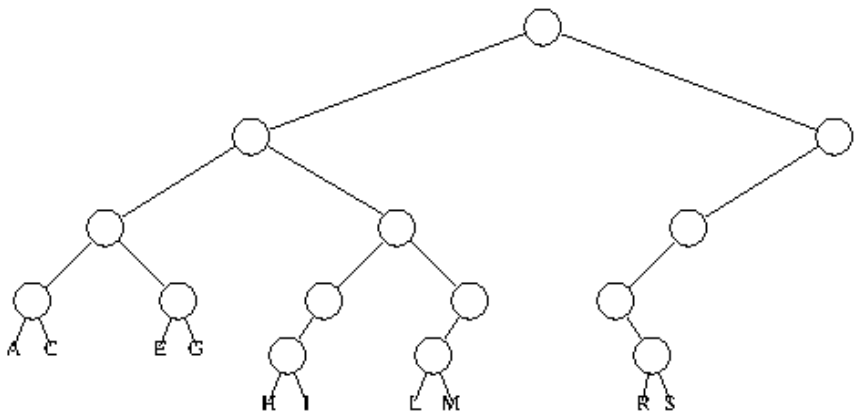
- *črke jemljemo po vrsti od prve naprej*
- *ob vsaki črki razdelimo množico na dva ($|\Sigma|$) podmnožic, dokler ni v podmnožici samo še en element*

- vstavljanje `J` $=$ `01010` je preprosto
- vstavljanje `O` $=$ `01111`: namesto `N` dodamo notranje vozlišče, ki ima lista `N` in `O`
- vstavljanje `Z` $=$ `11010`:
  - pri iskanju mesta naletimo na `X`
  - namesto `X` dodajamo nova notranja vozlišča, dokler ne dobimo vozlišča, kjer se `X` in `Z` razlikujeta



Brisanje

- postopek je obraten postopku vstavljanja
- brišemo `C`:
  - brišemo `C`
  - ker je notranje vozlišče nepotrebno, ga nadomestimo s preostalim listom `A`
- brišemo `X`, `Z`, `S`:
  - brišemo vse predhodnike `S`, ki imajo samo en element v poddrevesu
  - preostali element je lahko samo brat (zakaj?)



Analiza

- recimo, da je naš ključ velik m črk – v našem primeru je dolg 5 bitnih črk in od tu bomo analizirali primer, ko imamo samo binarno abecedo

Časovna analiza:

- v vsakem primeru potrebujemo $m = O(m)$ dostopov (primerjav), da se sprehodimo do lista;

Pri običajnem iskanju, kaj pa pri posplošenem iskanju?

Prostorska analiza:

- velikost strukture je v najslabšem primeru

$$2n - 1 + n(m - \lg n) = O(nm)$$

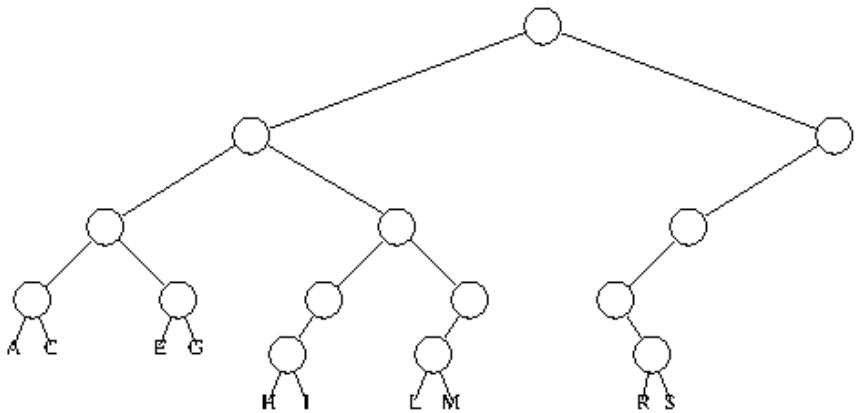
vozlišč. Zakaj?

Analiza – povzetek

- vse operacije imajo enak čas $O(m)$
- prostor je
$$2n - 1 + n(m - \lg n) = O(nm)$$
- kaj je najbolj motečega v naši strukturi (prostorsko)? Kakšne ideje za izboljšave?

Primer – poglejmo podrobneje

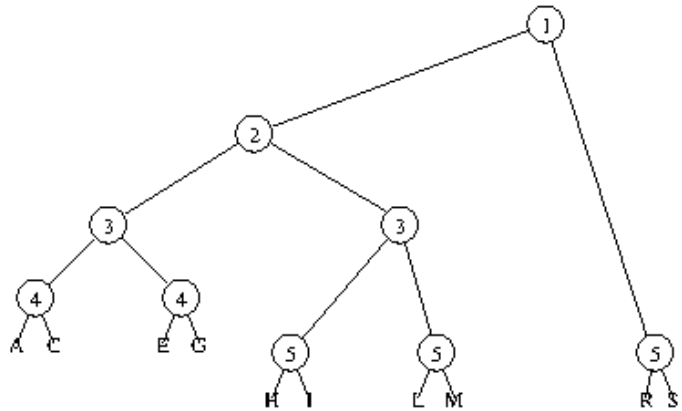
črka	*biti*	*črka*	*biti*
A	00001	C	00011
E	00101	G	00111
H	01000	I	01001
L	01100	M	01101
R	10010	S	10011



- POZOR: dostop do vsakega vozlišča stane eno enoto!
- koliko je: a) zunanjih vozlišč, b) notranjih vozlišč z enim naslednikom in c) notranjaih vozlišč z 2 naslednikoma.
- čemu potrebujemo vozlišča b) in čemu vozlišča c)? kakšna je razlika v uporabi enih in drugih?

Stiskanje poti – *path compression*

- na poti od korena do lista:
  - izpustimo vsa vozlišča, ki imajo samo enega naslednika
  - v preostala vozlišča dodamo informacijo, kateri bit po vrsti naj primerjamo ali koliko bitov naj preskočimo



OPAŽANJE: če smo izločili notranja vozlišča v, imajo vsi nasledniki tega vozlišča *poljubno vrednost* bita, ki ga je predstavljalo opuščeno vozlišče. Zato, ko pridemo do lista, smo našli *samo kandidata* in moramo še preveriti, ali smo našli iskani ključ ali katerega drugega.

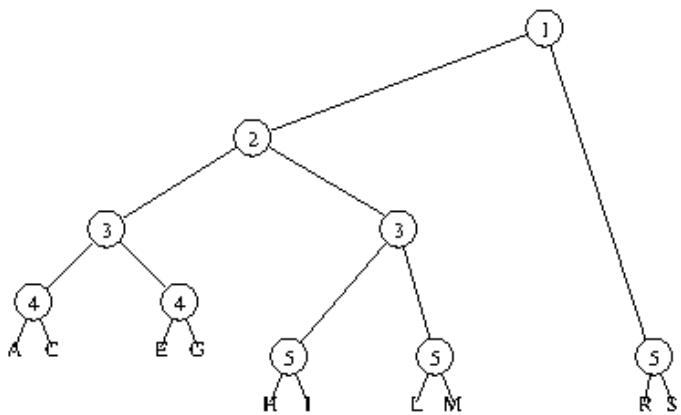
Stiskanje poti – analiza

čas: nespremenjen ali boljši

prostor: $O(n)$

Vstavljanje

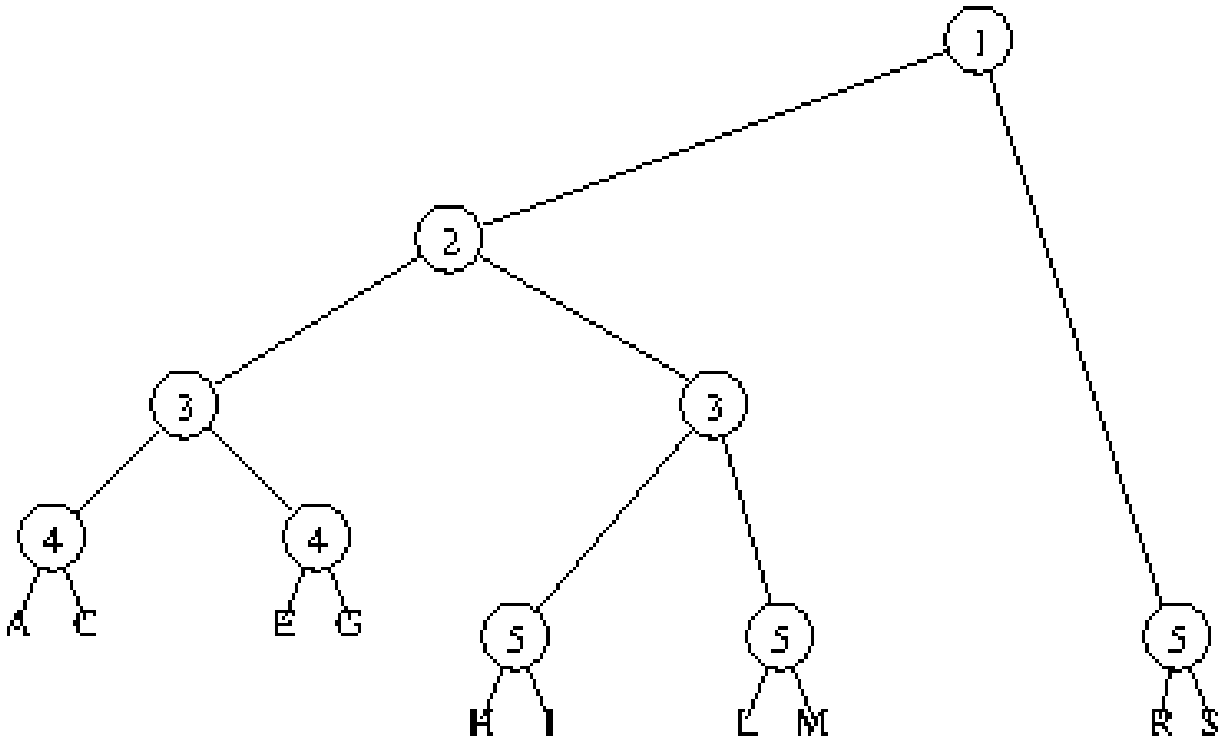
- OPAŽANJE: za vsak list velja, da indeksi bitov naraščajo po poti od korena do lista
- ko pridemo do lista, ki ne vsebuje vstavljanega elementa, se lahko zgodi:
  1. da se ključa elementov razlikujeta v bitu, ki je kasneje (nižje) od lista – tedaj samo vstavimo novo notranje vozlišče
  2. sicer je potrebno:
     - 2.1 poiskati prvi bit, na katerem se ključa razlikujeta (zakaj vozlišča za ta bit zagotovo še ni v strukturi?)
     - 2.2 na to mesto dodati novo notranje vozlišče (prim. zgornje opažanje)
     - 2.3 kot poddrevesi novega vozlišča nastopata staro poddrevo in list, ki predstavlja vstavljani element
- časovna zahtevnost ostaja $O(m)$



Vstavljanje – poenostavitev

Vedno nadomestimo list z notranjim vozliščem in indeksom bita, kjer se elementa razlikujeta ter stari in novi element postaneta nova lista.

Dodajmo: $X = 11000$. Pri tem sta: $R = 10010$ in $S = 10011$:



Kaj bo posledica? Je takšen pristop pravilen?

Vstavljanje – sprememba invariance

Stara invarianca:

- *črke jemljemo po vrsti od prve naprej (za vsak list velja, da indeksi bitov padajo po poti od korena do lista)*
- *ob vsaki črki razdelimo množico na dva ($|\Sigma|$) podmnožic, dokler ni v podmnožici samo še en element*

Nova invarianca (šibkejša):

- *za vsak list velja, da so indeksi bitov po poti od korena do lista različni*
- *ob vsaki črki razdelimo množico na dva ($|\Sigma|$) podmnožic, dokler ni v podmnožici samo še en element*

Brisanje ter iskanje soseda

- brisanje je obratna operacija vstavljanju, le da sedaj pobrišemo poleg lista še eno od notranjih vozlišč in sicer tisto, ki je prvi starš brisanemu listu
- časovna zahtevnost ostaja $O(m)$
- kaj v primeru *poenostavitve*?
- iskanje sosedov poteka na enak način kot pri običajnem trie
- kaj pa v primeru *poenostavitve*?

Polja in drevesa

- polja so *implicitne* podatkovne strukture, kjer do posameznih elementov dostopamo s pomočjo indeksa, ki je *izračunljiv* iz vrednosti ključa v času $O(1)$ – HITRO!!!;
- drevesa so *eksplicitne* podatkovne strukture, kjer do posameznih elementov dostopamo s pomočjo referenc – POČASI!!!;
- velikost drevesnih struktur je $O(n)$, oziroma toliko, da se shranijo vsi elementi in reference, ki jih pa ni (bistveno) več kot elementov;
- prostor, ki zaseda polje, načeloma *ni odvisen* od trenutnega števila elementov v strukturi n, ampak od števila vseh možnih elementov M – velikosti univerzalne množice.

Kako se temu izogniti? – NAMIG: amortizacijske podatkovne strukture.

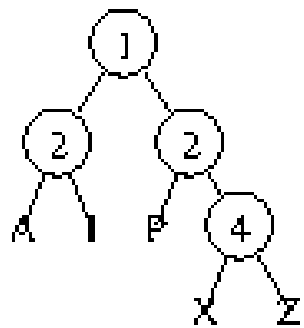
Najboljše od obeh

- ko je $n = O(M)$ (gosta množica), je smiselneje uporabiti polje in ko je n zelo majhen (redka množica), je smiselneje uporabiti drevo
- opažanje velja tudi za majhne delčke univerzalne množice – IDEJA:

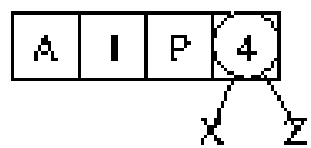
  uporabili bomo hkrati dve strukturi: polje in drevo, pač glede na lokalno gostoto

Primer

črka	*biti*	*črka*	*biti*
A	00001	I	01001
P	10000	X	11000
Z	11010		

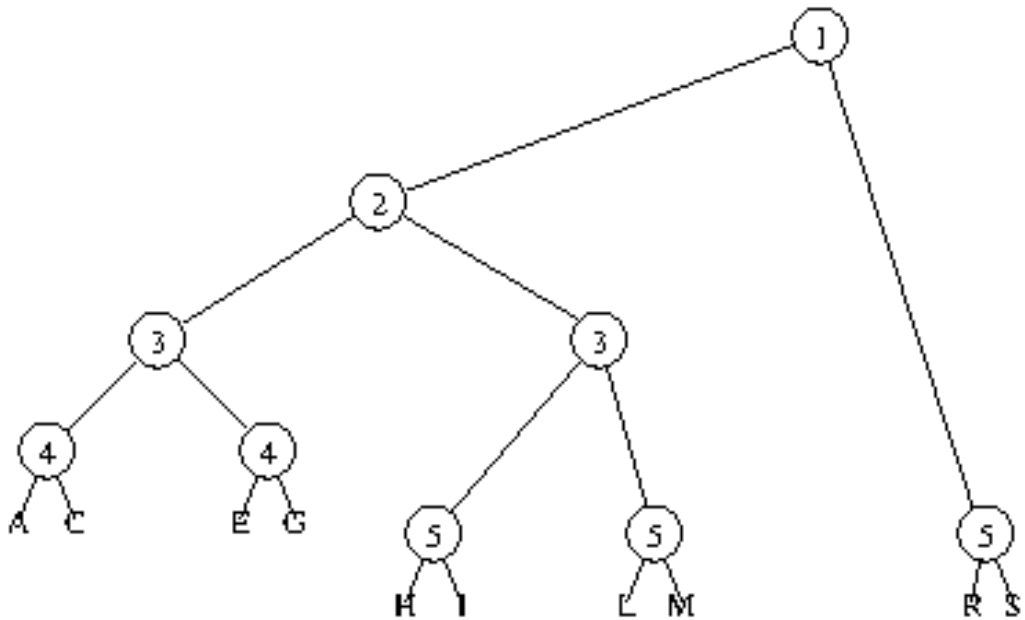


Zgornji nivoja predstavljata gosto množico in ju lahko nadomestimo s poljem ter dobimo:



Nivojsko stisnjena drevesa – *LC tries*

- definirajmo gostoto α in če je na nekem nivoju gostota elementov večja od α, drevo stisnemo še po nivoju
- postopek:
  1. pričnemo s drevesom PATRICIA
  2. od korena proti listom se spuščamo po nivojih (plasteh) l, dokler je v plasti več kot $\lfloor \alpha \cdot 2^l \rfloor$ elementov
  3. ko ni več, prejšnje plasti stisnemo in ponovimo korak 2 z novim korenom



Kako tvoriti optimalno LC drevo? – *optimizacijski problem*

Iskanje vzorca v besedilu

- imamo besedilo $T = \texttt{abracadabra}$.
- V njem iščemo vzorec $P_1 = \texttt{abrac} \rightarrow$ preprosto, ker je na začetku T.
- Kaj pa vzorec $P_2 = \texttt{dab}$? $\rightarrow$ Ker ni na začetku, ne izgled tako preprosto.
- Je pa preprosto v besedilu $T_7 = \texttt{dabra}$.

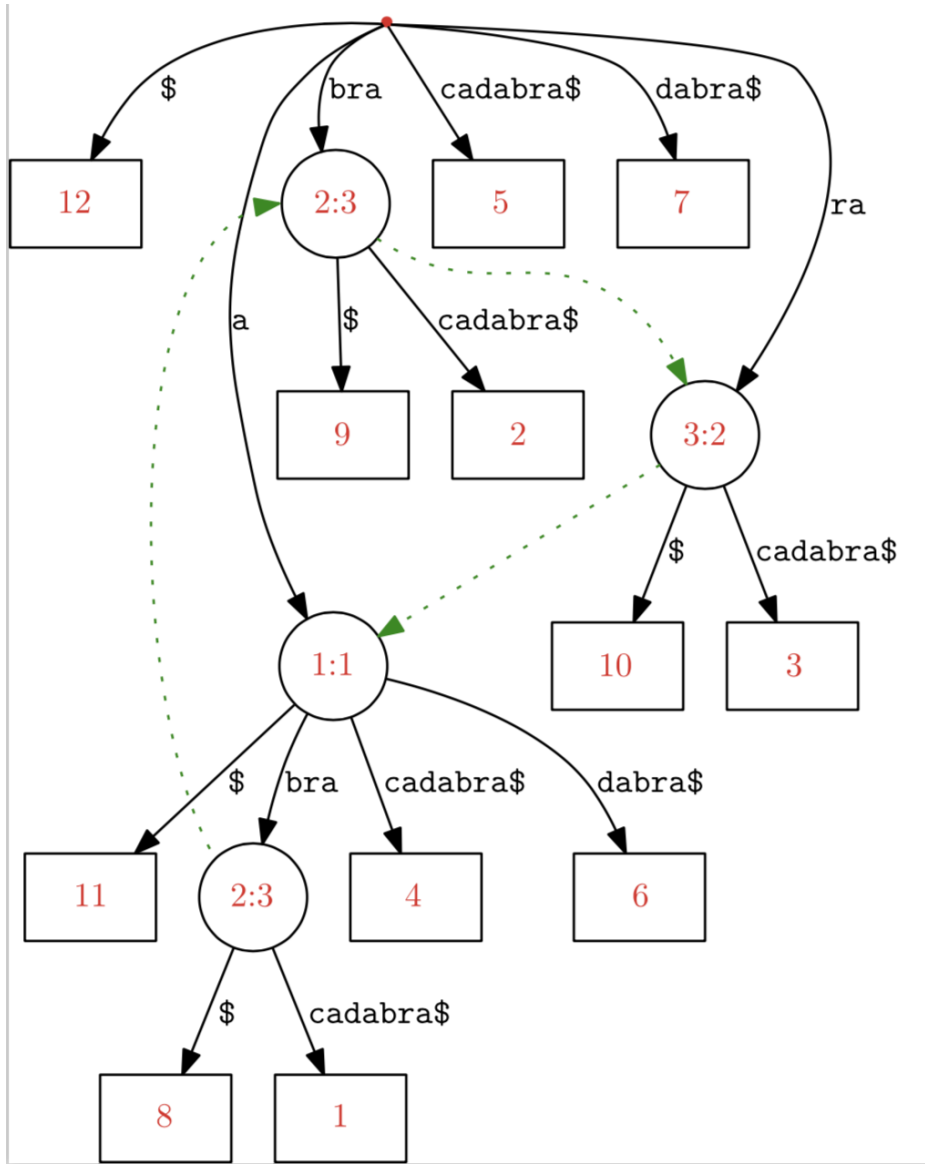
- T_7 je sedma *pripona* besedila T.
- Če bi imeli vse pripone $T_1 = T, T_2, T_3, \ldots T_{11}, T_{12} = \epsilon$, bi lahko poiskali katerikoli vzorec P v T tako, da bi našli pripono (ali pripone), na katere (katerih) začetku je P.
- Potrebujemo učinkovito podatkovno strukturo, ki hrani pripone in omogoča iskanje.

Številsko drevo in pripone

Tvorimo vse pripone:

$T = T_1 =$ `abracadabra`
$T_2 =$ `bracadabra`
$T_3 =$ `racadabra`
$T_4 =$ `acadabra`
$T_5 =$ `cadabra`
$T_6 =$ `adabra`
$T_7 =$ `dabra`
$T_8 =$ `abra`
$T_9 =$ `bra`
$T_{10} =$ `ra`
$T_{11} =$ `a`
$T_{12} = \epsilon$

in jih vstavimo v številsko drevo $\rightarrow$ *priponsko drevo*.

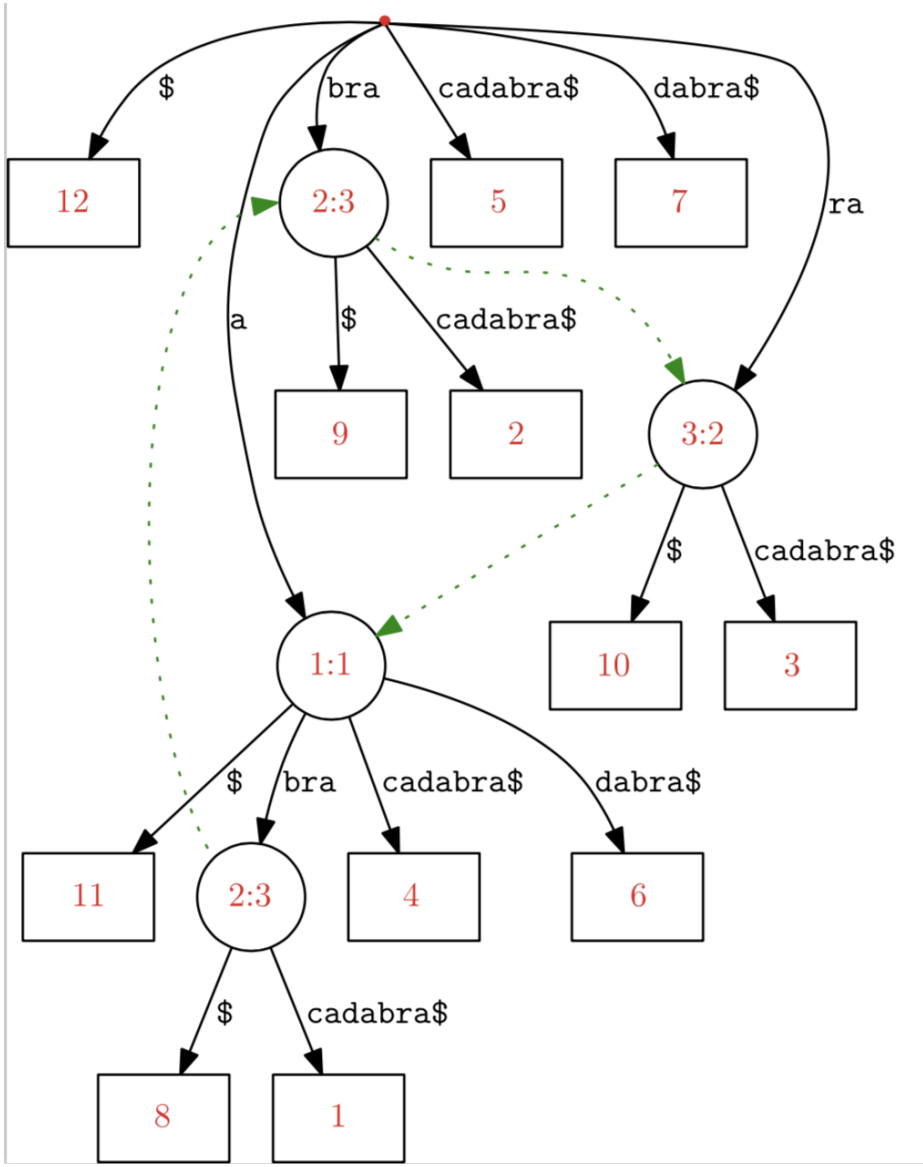


Priponsko drevo – analiza

časovna zahtevnost:

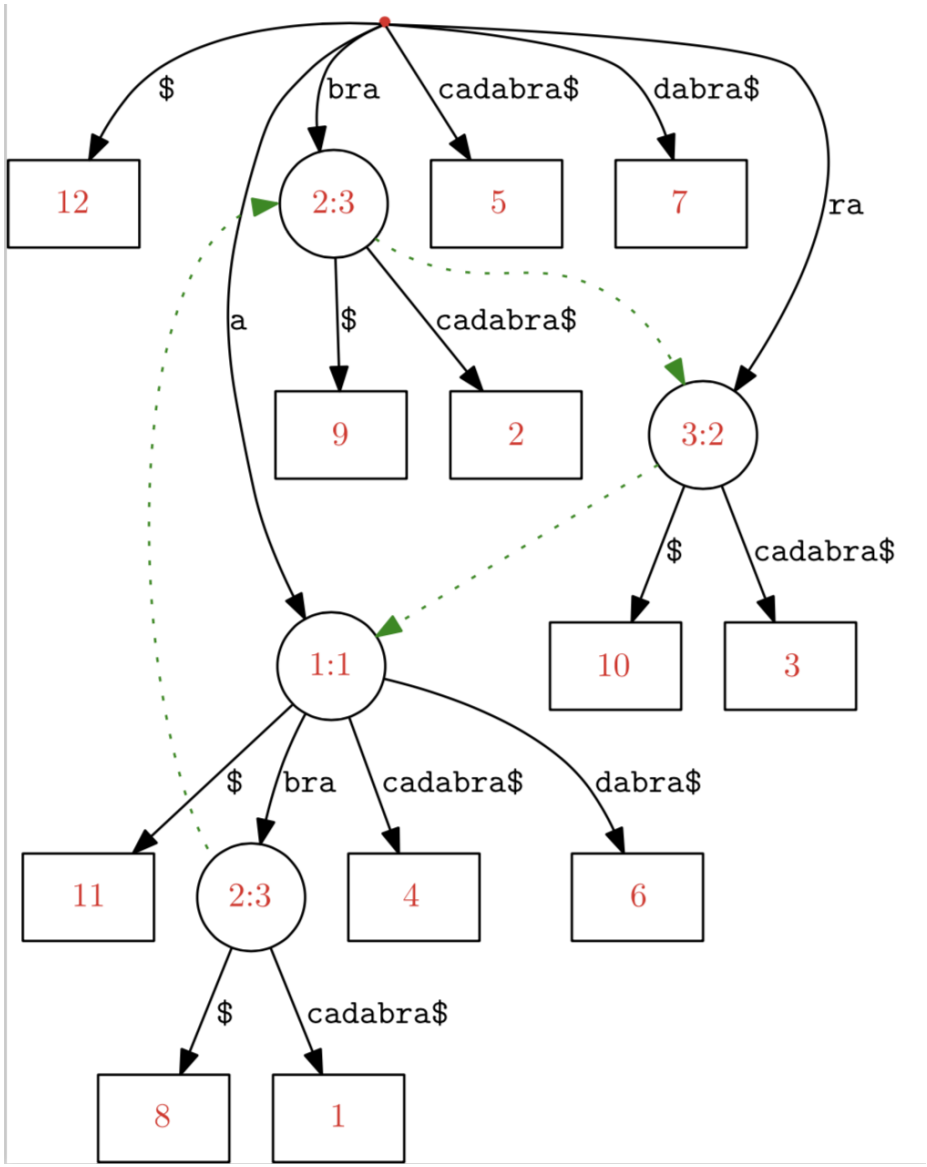
$O(|P| + \mathrm{occ})$, kjer je occ število pojavitev vzorca P v besedilu T.

prostorska zahtevnost: podatki: n, reference $3n$



Razširitev, da učinkovito najdemo vsa mesta P v T.

Drevesa in polja



$T =$

1	a
2	b
3	r
4	a
5	c
6	a
7	d
8	a
9	b
10	r
11	a
12	$

$S =$

1	12
2	11
3	8
4	1
5	4
6	6
7	9
8	2
9	5
10	7
11	10
12	3