Algoritmi in podatkovne strukture – 2

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 `re`**`trie`**`val` (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

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

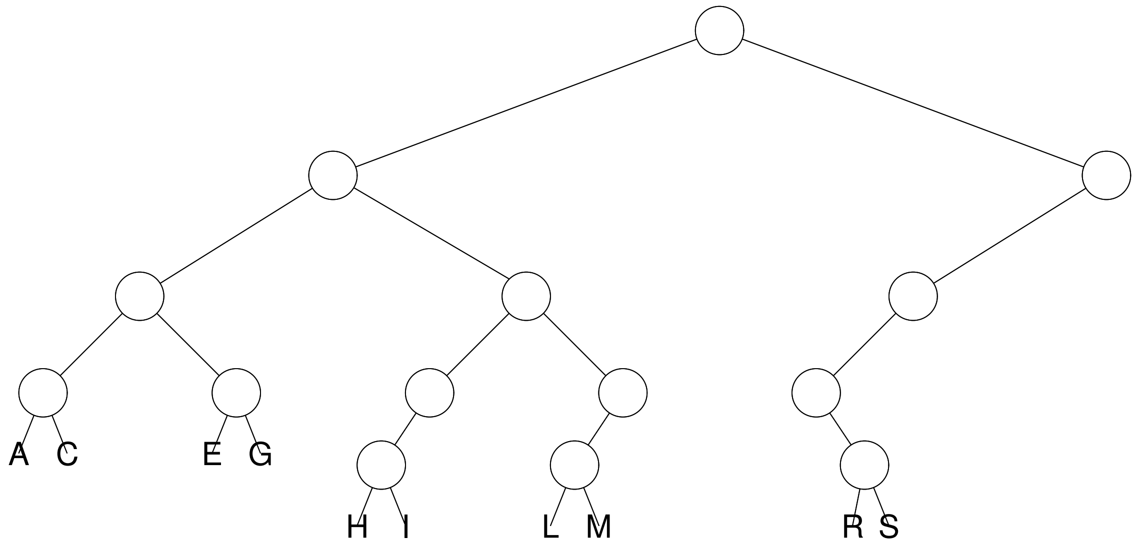
Gradnja

– 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 levega soseda, ali najmanjšega elementa v strukturi, ki je že večji od iskanega elementa:

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

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*

- 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?)

Brisanje – algoritem

```
pobriši element
if (brat IS list)
  zamenjaj starša z bratom
  while (stari starš HAS samo enega otroka)
    samenjaj starega starša s staršem
```

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
- v vsakem primeru potrebujemo $m = O(m)$ dostopov (primerjav), da se sprehodimo do lista pri običajnem iskanju, kaj pa pri posplošenem iskanju?
- velikost strukture je v najslabšem primeru

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

  vozlišč. Zakaj?
- največ lahko rokujemo z $M = 2^m$ različnimi elementi, ki jim pravimo *univerzalna množica*. Primeri univerzalnih množic:
  - naše črke-ključi tvorijo množico velikosti $2^5 = 32$
  - vsi IPv4 naslovi na svetu tvorijo množico $2^{32} = 4.294.967.296$ IP naslovov
  - vsa cela števila, s katerimi (običajno) računamo, tvorijo množico $2^{64} = 18.446.744.073.709.551.616$ števil, ...

- cela števila lahko obravnavamo kot elemente z enovitim ključem, ali kot elemente s ključem sestavljenim iz črk (binarne/bitne) abecede – npr. namesto po bitih, po zlogih
- na primer, ko n postaja vedno večji ter se po velikosti približuje M, postane $O(m)$ iskanje povsem primerljivo z iskanji v običajnih podatkovnih strukturah
- Vprašanje: imamo univerzalno množico velikosti M in njeno podmnožico velikosti $n < M/2$. Koliko bitov potrebujemo, da predstavimo podmožico?

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

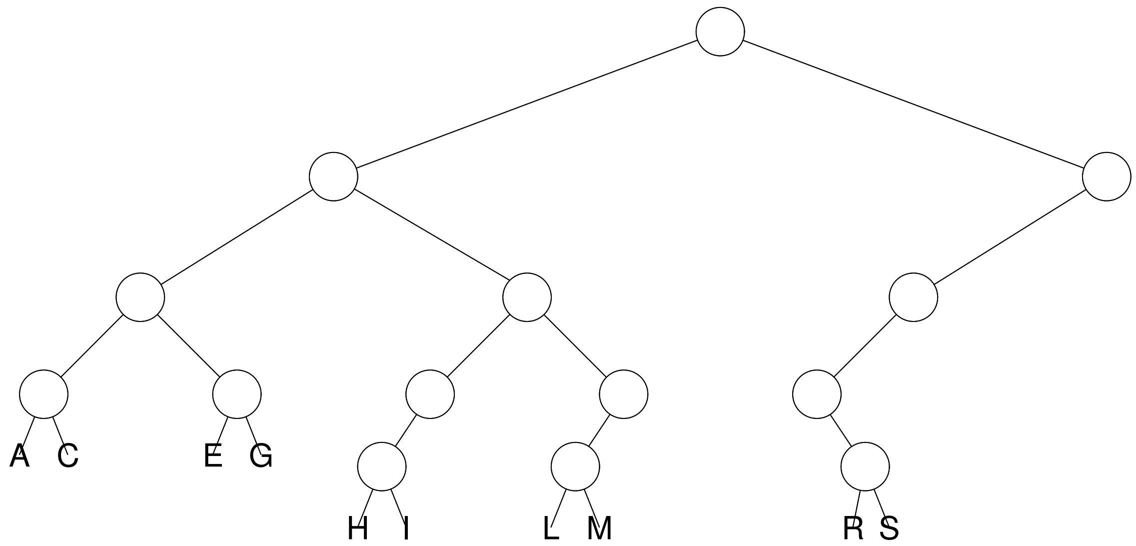
- imamo binarne (bitne) črke abecede $\{0, 1\}$ in imejmo bitno predstavitev naslednjih ključev:

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

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

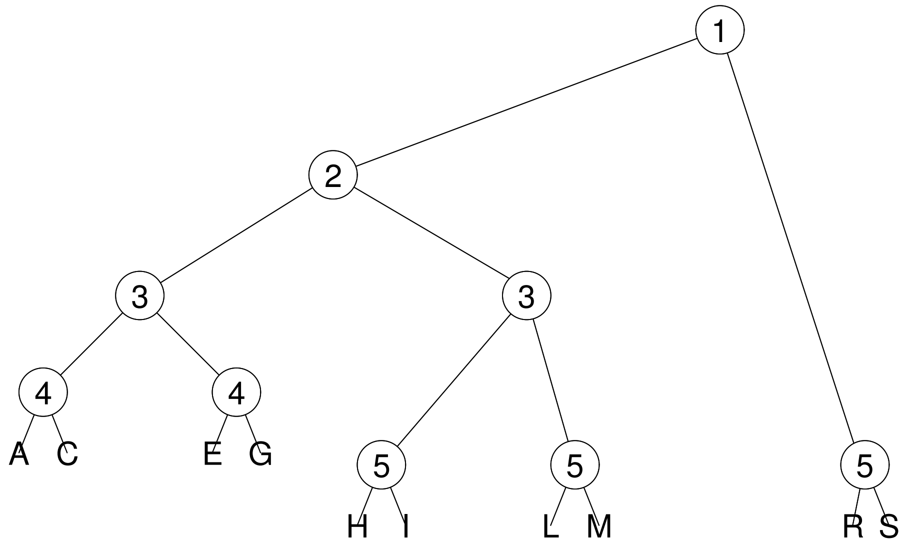
Primer



- opazujmo predvsem naslednje tri vrste vozlišč: a) zunanja, b) notranja z enim naslednikom in c) notranja z 2 naslednikoma.
- POZOR: dostop do vsakega vozlišča stane eno enoto!
- kolikšno je število vozlišč a), b) in c)?
- čemu potrebujemo vozlišča b) in čemu vozlišča c)? kakšna je razlika v uporabi enih in drugih?
- ugotovitve so: ...

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

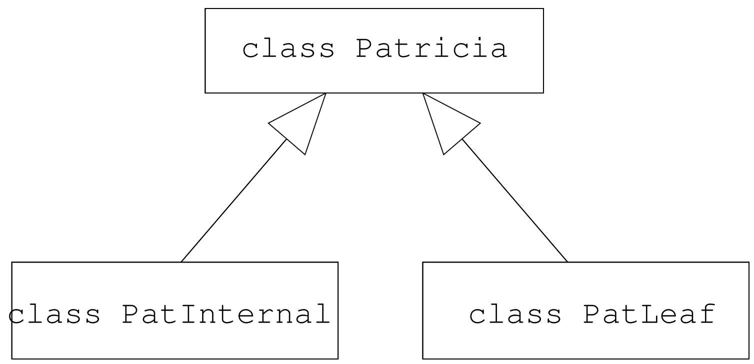
čas: nespremenjen ali boljši
prostor: $O(n)$

Definicija razreda

- imamo dve vrsti vozlišč:

  notranja: hranijo podatke o levem in desnem poddrevesu ter številko primerjanega bita

  listi: hranijo element
- vsa vozlišča pa imajo enake metode: iskanje, vstavljanje in izločanje
- torej naredimo vmesnik z definicijo vseh teh metod in iz njega izpeljemo dva različna razreda
- definirajte vmesnik in oba razreda za domačo nalogo



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*

- OPAŽANJE: za vsak list velja, da indeksi bitov padajo 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:
     (a) poiskati prvi bit, na katerem se ključa razlikujeta (zakaj vozlišča za ta bit zagotovo še ni v strukturi?)
     (b) na to mesto dodati novo notranje vozlišče (prim. zgornje opažanje)
     (c) kot poddrevesi novega vozlišča nastopata staro poddrevo in list, ki predstavlja vstavljani element
- časovna zahtevnost ostaja $O(m)$

Vstavljanje – poenostavitev

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*

Vedno dodamo notranje vozlišče na mestu lista, le v tem primeru zgornje opažanje ne bo več držalo. Kaj bo posledica? Je takšen pristop pravilen?

Nova invarianca:

- *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 $O(1)$ času
- drevesa so *eksplicitne* podatkovne strukture, kjer do posameznih elementov dostopamo s pomočjo referenc
- 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 (ima kdo kakšno idejo, kako se temu izogniti? – NAMIG: amortizacijske podatkovne strukture.)

- v splošnem, če imamo univerzalno množico velikosti M in iz nje n elementov, potem:
  polje: potrebuje $O(M)$ prostora in $O(1)$ časa ter
  drevo: potrebuje $O(n)$ prostora in $O(\log n)$ ($O(\log m)$) časa
  optimalno: potrebujemo najmanj

$$\lg \left\lceil \binom{M}{n} \right\rceil \approx n \lg \frac{M}{n} = n(\lg M - \lg n) = n(m - \lg n)$$

  bitov

Najboljše od obeh

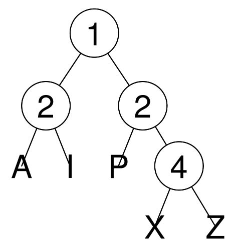
- ko je $n = O(M)$ (gosta množica), je smiselneje uporabiti polje in ko je n zelo majhen (redka množica), je smiselneje uporabiti drevo
- opažanje velja tudi za majhne delčke univerzalne množice – IDEJA:

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

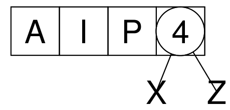
Primer

Vzemimo, da imamo naslednje elemente

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



Zgornji nivoji predstavljajo gosto množico in jih 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

IZZIV: kako tvoriti optimalno LC drevo? – optimizacijski problem