Cvetana Krstev

Strukture informacija – deo 4

Apstraktni tipovi podataka

Apstraktni tipovi podataka, za razliku od konstruisanih tipova, ne mogu se izraziti neposredno u klasičnim programskim jezicima. Apstraktni tipovi, po pravilu, ne zavise od programskog jezika, kao ni od načina na koji će biti prikazani u određenom jeziku. Neki apstraktni tipovi o kojima ćemo govoriti su *sekvencijalne liste*, *drveta* i *grafovi*.

Sekvencijalne liste

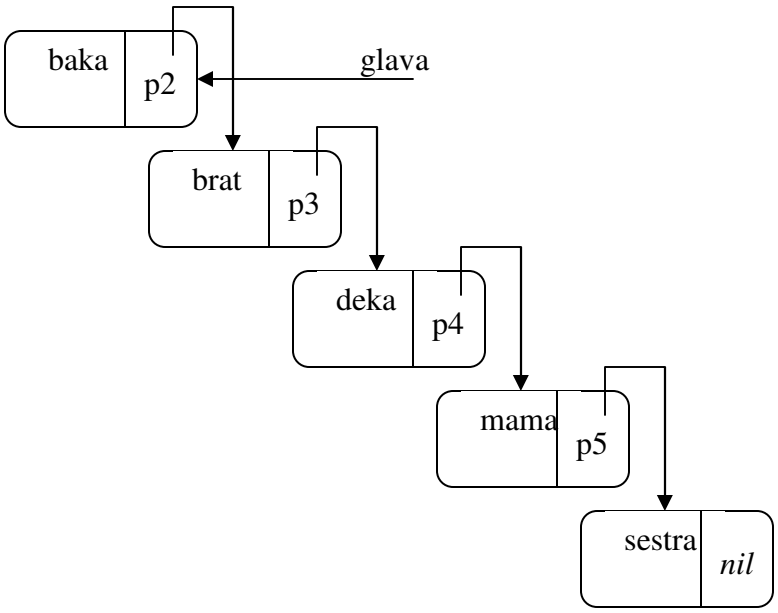
Sekvencijalna (ili *jednostruka* ili *linerana*) *lista* predstavlja niz elemenata određenog tipa **T** kome su pridružene posebne operacije pristupa određenom elementu. Pristup elementima liste je strogo *sekvencijalan*: ako su e_1, e_2, e_3,..., e_n elementi tipa **T**, onda je sekvencijalna lista $\mathbf{L} = (e_1, e_2, e_3,..., e_n)$ pri čemu je $n \geq 0$. Broj elemenata liste n nazivamo *dužina liste*. Ako je $n = 0$ kažemo da je lista *prazna* i obeležavamo je sa ε. Obilazak liste podrazumeva da je poznato:

(a) gde lista počinje (prvi element liste e_1) preko posebne promenljive koju nazivamo *glava liste* i obeležavamo sa *glava*(L); i
(b) funkcija *naslednik* (obeležavamo je strelicom →) kojom se određuje *sledeći element* liste:
glava liste(L) → e_1→ e_2→ e_3→...→ e_n

Element liste e_i zauzima i-tu poziciju, kojoj odgovara adresa p_i u memoriji. Sekvencijalni prilaz elementima liste se izražava sledećim uslovima:

glava liste(L) = p_1
naslednik(e_i) = p_{i+1}

Uslov koji neki element e tipa **T** treba da zadovolji da bi bio element liste je da postoji put od glave liste do elementa e sledeći funkciju naslednik. Funkcija naslednik nije definisana za poslednji element liste e_n, tj. *naslednik*(e_n) = *nil*.



Na slici je vizuelno prikazana jedna sekvencijalna lista koja se sastoji od 5 elemenata e_1, e_2, e_3, e_4, e_5. Svaki element liste sadrži jednu nisku karaktera – tip **T** elemenata liste je u ovom slučaju `String` (npr. u jeziku VBA). Promenljiva *glava* pokazuje prvi element liste, tj. element od koga lista počinje. Na osnovu ove slike se vidi da se svaki čvor u listi sastoji, u stvari, od dva podatka: od sadržaja koji je tipa **T** i od *pokazivača* koji pokazuje naredni element liste. Prilikom realizacije liste taj pokazivač zapravo sadrži adresu u memoriji na kojoj se nalazi čvor narednog elementa liste. Prema tome, ova dva podatka zajedno čine jedan *zapis* (eng. record). To bismo mogli da predstavimo na sledeći način:

zapis *elementListe*
    reč **Niska,**
    sledeći ↑*elementListe*
kraj zapisa

Promenljiva *reč*, kao deo zapisa *elementListe*, je sadržaj elementa liste (promenljiva tipa **T**), dok strelica nagore u ovom zapisu govori da je promenljiva *sledeći*, kao deo zapisa *elementListe*, pokazivač (tj. adresa), nekog zapisa *elementListe*. U svim narednim primerima ćemo pretpostavljati da imamo definisan ovakav zapis. Tada bi algoritam koji prikazuje redom sve elemente liste bio:

„Prikaži sve elemente liste“
e, glava su pokazivači elementa liste - ↑*elementListe*
e ← *glava*
sve dok je *e* <> nil
    prikaži *e*↑.*reč*
    e ← *e*↑.*sledeći*
do ovde

U ovom algoritmu *e*↑ je čvor liste (tj. zapis *elementListe*) koji se nalazi na adresi *e*. Ako bi se ovaj algoritam primenio na listu sa prethodne slike dobili bismo:

Glava	adr(e_1)					
e	adr(e_1)	adr(e_2)	adr(e_3)	adr(e_4)	adr(e_5)	*nil*
e <> nil	T	T	T	T	T	F
e↑.*reč*	baka	brat	deka	mama	sestra	

☺

Algoritam koji utvrđuje da li se neka vrednost *x* koja je takođe tipa **T** nalazi u listi bio bi:

„Utvrdi da li je element *x* u listi“
e, glava su pokazivači elementa liste - ↑*elementListe*
x je niska (**Niska**)
e ← *glava*
x nije u listi
sve dok je *e* <> nil **i** *x* nije u listi
    ako je *e*↑.*reč* = *x* **onda**
        x je u listi
    inače
        e ← *e*↑.*sledeći*
    kraj ako-onda
do ovde

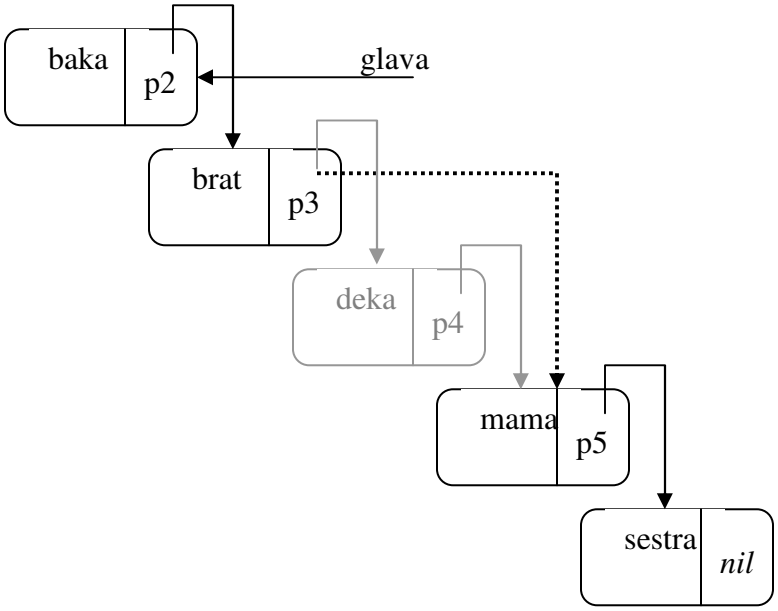
Ako bi se ovaj algoritam primenio na listu sa prethodne slike i ako bismo u njoj tražili nisku „mama“ dobili bismo:

Ulaz: Lista i x = „mama“

Glava	adr(e_1)				
e	adr(e_1)	adr(e_2)	adr(e_3)	adr(e_4)	isto
x nije u listi	T	T	T	T	F
e <> nil	T	T	T	T	T
uslov	T	T	T	T	F
e↑.*reč*	baka	brat	deka	mama	
e↑.*reč* = *x*	F	F	F	T	

Odgovor: x je u listi, element e_4

☺



Pretpostavimo sada da iz liste želimo da isključimo neki element – na primer, želimo iz liste sa gornje slike da isključimo element e_3 koji sadrži reč „deka“. Setimo se da je u slučaju niza izbacivanje elementa podrazumevalo pomeranje svih članova niza koji su dolazili iza njega (čiji je indeks u nizu bio veći). U slučaju sekvencijalne liste ništa se ne pomera, potrebno je samo da se podese pokazivači: pokazivač elementa koji prethodi onom koji se izbacuje treba da pokaže kao sledeći element onaj koji sledi element koji se izbacuje. Element koji se izbacuje ne mora stvarno i da se briše iz memorije – on jednostavno više nije u listi jer se do njega nikako ne može doći polazeći od glave liste i sledeći pokazivače ka sledećem elementu liste.

„Izbaci element *x* iz liste, ako je u njoj“
e, glava, stari su pokazivači elementa liste - ↑*elementListe*
x je niska (**Niska**)
e ← *glava*
x nije u listi
ako je *e*↑.*reč* = *x* **onda** /* prvi element liste je onaj koji tražimo */
    glava ← *e*↑.*sledeći*
inače /* element koji tražimo nije prvi element liste */
    stari ← *glava*
    e ← *e*↑.*sledeći*
    sve dok je *e* <> nil **i** *x* nije u listi
        ako je *e*↑.*reč* = *x* **onda**
            stari↑.*sledeći* ← *e*↑.*sledeći*
            x je u listi
        inače
            stari ← *e*
            e ← *e*↑.*sledeći*
        kraj ako-onda
    do ovde
kraj ako-onda

Ako bi se ovaj algoritam primenio na polaznu listu i ako bismo iz nje isključili nisku „deka“ algoritam bi funkcionisao na sledeći način:

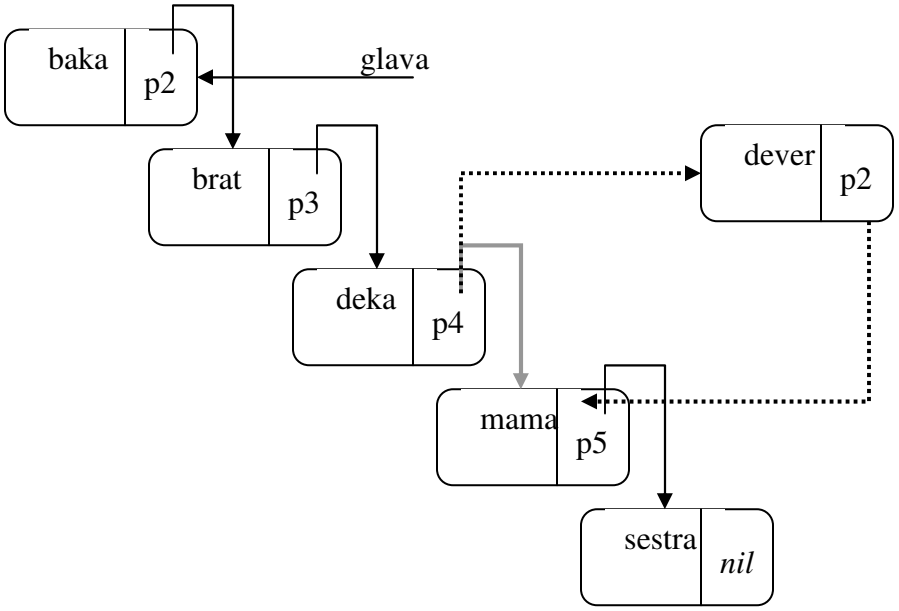
Ulaz: Lista i *x* = „deka“

e↑.*reč* baka	*e*↑.*reč* = *x* F		
e	adr(e_2)	adr(e_3)	Isto
stari	adr(e_1)	adr(e_2)	Isto
e <> nil	T	T	T
x nije u listi	T	T	F
Uslov	T	T	F
e↑.*reč*	brat	deka	
e↑.*reč* = *x*	F	T	
stari↑.*sledeći*		adr(e_4)	

☺

Pretpostavimo sada da u listu želimo da dodamo neki element. Setimo se da je u slučaju niza dodavanje elementa podrazumevalo pomeranje svih članova niza koji su dolazili iza pozicije na kojoj se novi element dodaje (čiji

je indeks u nizu bio veći). U slučaju sekvencijalne liste ništa se ne pomera, potrebno je samo da se kreira novi element liste, a zatim da se podese pokazivači: pokazivač elementa koji prethodi novom elementu treba da pokazuje kao sledeći upravo taj novi element, a pokazivač novog elementa treba da pokaže kao sledeći onaj element koji je njegov prethodnik ranije pokazivao kao sledeći. Na primer, želimo u uređenu listu od 5 reči: baka, brat, deka, mama, sestra da dodamo novi element koji će sadržati reč „dever“, ali tako da lista ostane uređena.



„Ubaci element *x* u uređenu sekvencijalnu listu“
e, glava, stari, *novi* su pokazivači elementa liste - ↑*elementListe*
x je niska (**Niska**)
e ← *glava*
noviElementListe(*novi*)
x nije dodat
ako je *e*↑*.reč* > *x* **onda** /* novi element dolazi na početak liste */
    novi↑*.reč* ← *x* /* npr. element *baba* */
    novi↑*.sledeći* ← *glava*
    glava ← *novi*
inače /* novi element ne dolazi na početak liste */
    stari ← *glava*
    e ← *e*↑*.sledeći*

```
        sve dok je e <> nil i x nije dodat
            ako je e↑.reč > x onda
                stari↑.sledeći ← novi
                novi↑.reč ← x
                novi↑.sledeći ← e
                x je dodat
            inače
                stari ← e
                e ← e↑.sledeći
            kraj ako-onda
        do ovde
        ako je x nije dodat
            /* stigli smo do kraja liste – novi treba dodati na kraj */
            /* npr. element ujak */
            stari↑.sledeći ← novi
            novi↑.reč ← x
            novi↑.sledeći ← nil
        kraj ako-onda
    kraj ako-onda
```

Ulaz: Lista i x = „dever“

početak	e↑.*reč* baka		*e*↑.*reč* > *x* F	
Ponavljanje Sve dok je	e	adr(e_2)	adr(e_3)	adr(e_4)
	stari	adr(e_1)	adr(e_2)	adr(e_3)
	e <> nil **i** *x* nije dodat	T	T	T
	e↑.*reč*	brat	deka	mama
	e↑.*reč*> *x*	F	F	T
kraj (u **ako-onda**)	*novi*↑.*reč* dever		*novi*↑.*sledeći* adr(e_4)	*stari*↑.*sledeći* adr(*novi*)

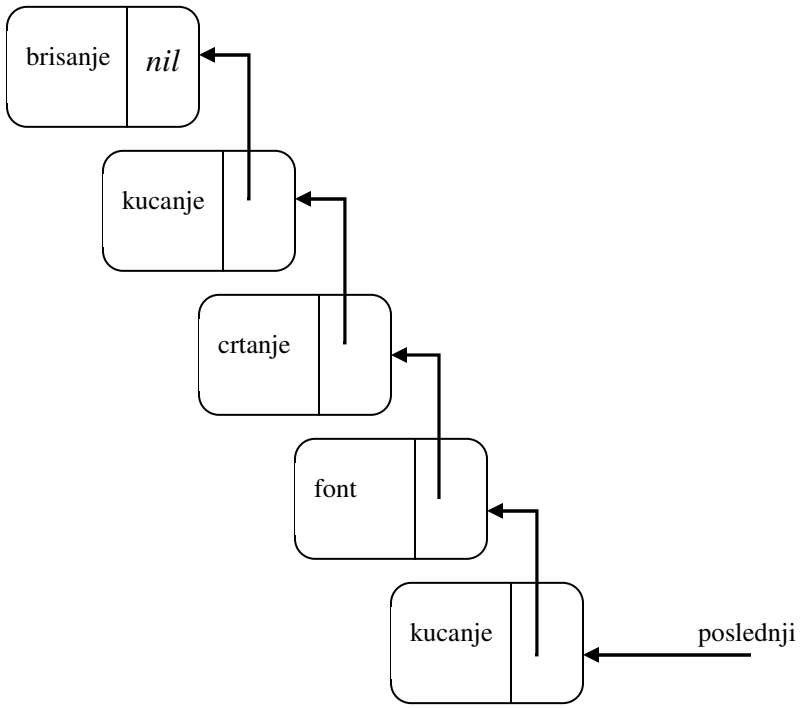
☺

LIFO liste

Posebna vrsta liste su takozvane *LIFO* liste čiji naziv potiče od engleskog *Last-In First-Out*, a ovakva lista se ponekad naziva i *stog*, od engleskog *stack*. Ovaj naziv potiče od posebnog načina pristupa elementima liste. Element liste se uvek dodaje na njen vrh, tj. iznad elementa koji je poslednji

dodat listi, a iz liste se briše uvek samo element koji je na njenom vrhu, to jest koji je poslednji dodat listi. Otuda i naziv *stog*, jer se seno na stog uvek dodaje na njegov vrh, a tako se i skida s njega. Slično je i sa tanjirima koji se slažu u kredenac: oprani tanjiri se dodaju na vrh, kada ih treba upotrebiti skidaju se s vrha. Na principu ovakve liste funkcionišu i funkcije *Undo* i *Redo*, recimo u programu Word: svaka nova akcija učinjena tokom pripreme teksta se pamti na vrhu za to predviđene liste (stoga), a kada nismo zadovoljni rezultatom neke akcije funkcijom *Undo* skidaju se akcije sa stoga i poništava njihov efekat, ali obrnutim redom od onoga kojim su obavljene – poslednja učinjena akcija se poništava prva.

Da bi ovakav pristup LIFO listi bio moguć, potrebno je da se u svakom čvoru liste, osim podatka, pamti i pokazivač (adresa) prethodnog čvora, a osim toga posebna promenljiva treba da pamti adresu poslednjeg dodatog čvora – vrh LIFO liste. Izgled jedne LIFO liste je prikazan na sledećoj slici.



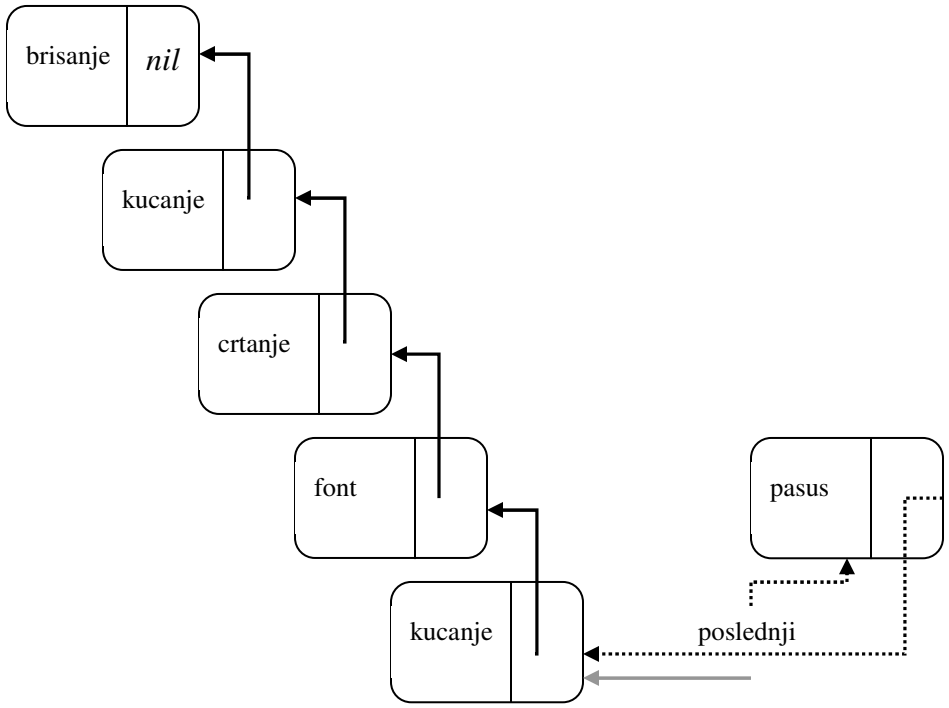
Zapis kojim bismo opisali čvor jedne LIFO liste i promenljiva koja označava njen vrh bi mogli da se opišu ovako:

poslednji ↑*elementLIFO*
zapis *elementLIFO*
reč **Niska,**

prethodni ↑*elementLIFO*
kraj zapisa

S obzirom da se LIFO listi može pristupiti samo na jedan način, preko vrha liste, za rad sa njom su potrebne samo dve operacije: dodavanje čvora na listu i skidanje čvora s liste.

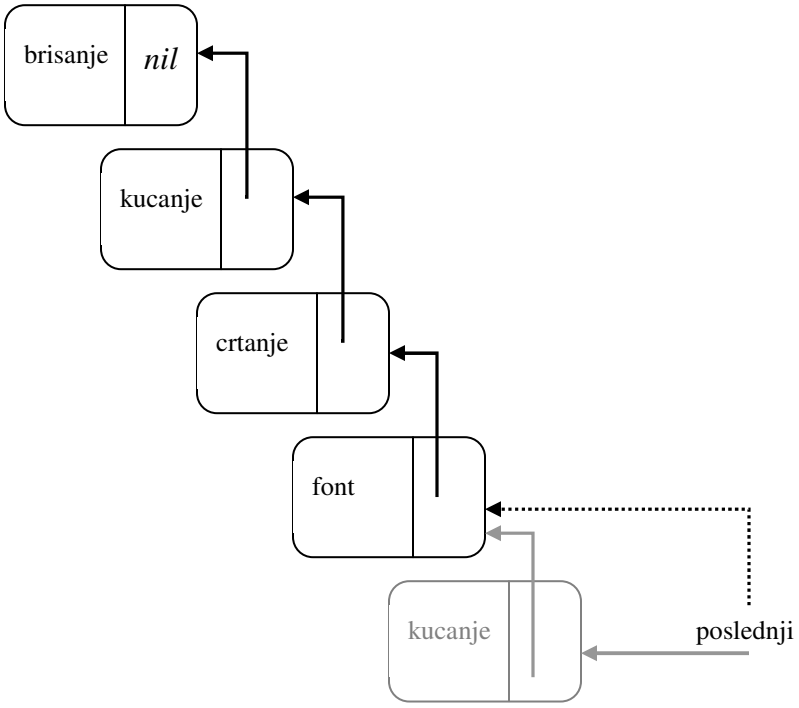
Da bi se dodao element na vrh LIFO liste, treba da se kreira novi čvor, taj čvor treba da se poveže sa prethodnim vrhom liste, a to je onaj element koga pokazuje promenljiva *poslednji*, a zatim treba promeniti i sadržaj promenljive *poslednji* koja sada treba da pokazuje novi vrh liste. Na sledećoj slici je prikazano kako bi izgledala naša LIFO lista kada bi korisnik Worda, pošto je nešto brisao, kucao, crtao, birao font, opet kucao, izabrao opciju formatiranja pasusa. Tada bi i ta poslednja akcija trebalo da se doda na listu obavljenih akcija.



„Dodaj element *x* na vrh LIFO liste“
poslednji, novi su pokazivači elementa liste - ↑*elementLIFO*
x je niska (**Niska**)
noviElementListe(*novi*)
novi↑*.reč* ← *x*
novi↑*.prethodni* ← *poslednji*
poslednji ← *novi*
☺

U gornjem algoritmu nije potrebna nikakva posebna akcija u slučaju kada je LIFO lista prazna, tj. vrednost promenljive *poslednji* je *nil*.

Da bi se uklonio element sa vrha LIFO liste, treba da se pročita sadržaj čvora koji je na vrhu liste, a to je onaj element koga pokazuje promenljiva *poslednji*, a zatim treba promeniti i sadržaj promenljive *poslednji* koja sada treba da pokazuje element koga je do tada pokazivao element s vrha liste koga sada skidamo. Na sledećoj slici je prikazano kako bi izgledala naša LIFO lista kada bi korisnik Worda, pošto je nešto brisao, kucao, crtao, birao font, opet kucao, odlučio da poništi poslednju akciju kucanja. Kao i ranije, izbačeni čvor ne mora da se briše iz memorije računara; on više nije povezan s listom pa se do njega nikako ne može ni doći.



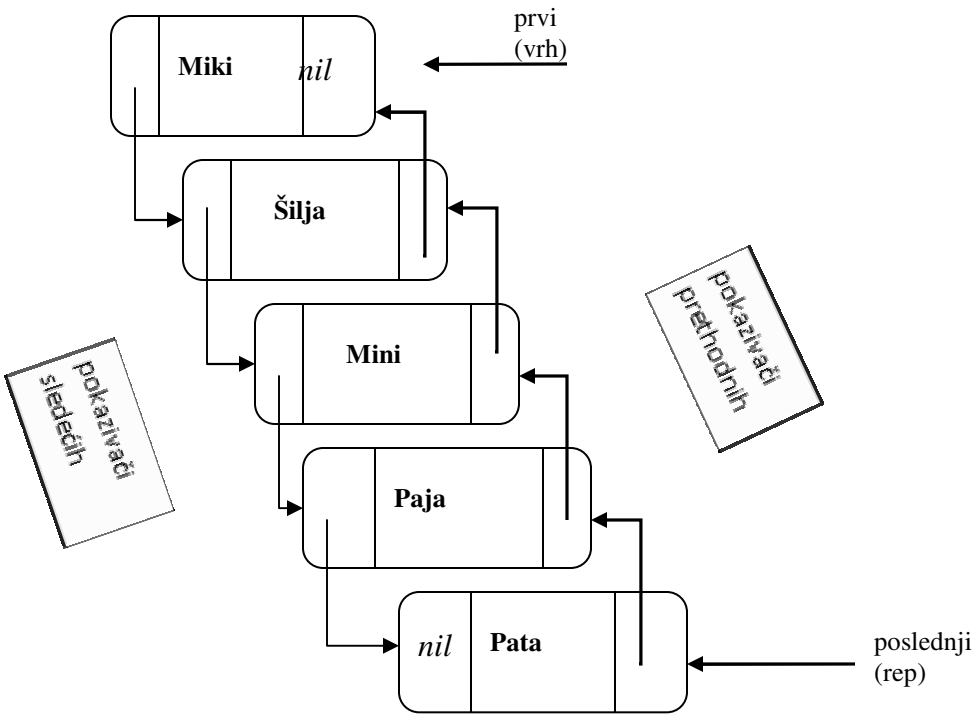
„Skidaj element *x* sa vrha LIFO liste“
poslednji je pokazivač elementa liste - ↑*elementLIFO*
x je niska (**Niska**)
ako *poslednji* <> nil **onda**
    x ← *poslednji*↑*.reč*
    poslednji ← *poslednji*↑*.prethodni*
kraj ako-onda
☺

Ni u ovom slučaju nije potrebna nikakva posebna akcija ako je LIFO lista prazna, samo što u tom slučaju neće moći da se pročita ništa s liste (promenljiva *x* ostaje nedefinisana).

FIFO

Posebna vrsta liste su takozvane *FIFO* liste čiji naziv potiče od engleskog *First-In First-Out*. Ovaj naziv potiče od posebnog načina pristupa elementima liste. Element liste se uvek dodaje na njen kraj ili rep, tj. iza elementa koji je poslednji dodat listi, a iz liste se briše uvek samo element koji je na njenom vrhu, to jest koji je prvi dodat listi (od onih koji su još u njoj). Ova lista se, prema tome, puni na jednom kraju a prazni na drugom. Na principu ovakve liste funkcionišu, na primer redovi ispred šaltera: svaki novi klijent staje na kraj reda, a kada se šalter oslobodi klijent koji je na vrhu izlazi iz reda i pristupa šalteru.

Da bi ovakav pristup FIFO listi bio moguć, potrebno je da se u svakom čvoru liste, osim podatka, pamti i pokazivač (adresa) prethodnog čvora. Ovaj pokazivač nalikuje uobičajenom ponašanju u redovima kod lekara. Tamo pacijenti ne stoje u redu, već se slobodno smeštaju u čekaonicu, ali svaki novi pacijent pita sve prisutne „Ko je poslednji?“ – kada dobije odgovor, novodošli se povezuje sa starim repom liste, a sam postaje rep. U svakom čvoru se osim toga pamti i pokazivač sledećeg čvora u listi, na primer, da bi uvek bilo jasno ko će sledeći doći na red. Zato se ove liste zovu i *dvostruko povezane liste*. Zbog svega toga su potrebne i dve posebne promenljive koje pamte adresu poslednjeg dodatog čvora – rep FIFO liste i adresu prvog dodatog čvora – vrh FIFO liste. Izgled jedne FIFO liste (recimo red ispred šaltera u banci) je prikazan na sledećoj slici. U ovom slučaju Miki će biti prvi uslužen, a Pata je poslednji stigla i stala na kraj reda.

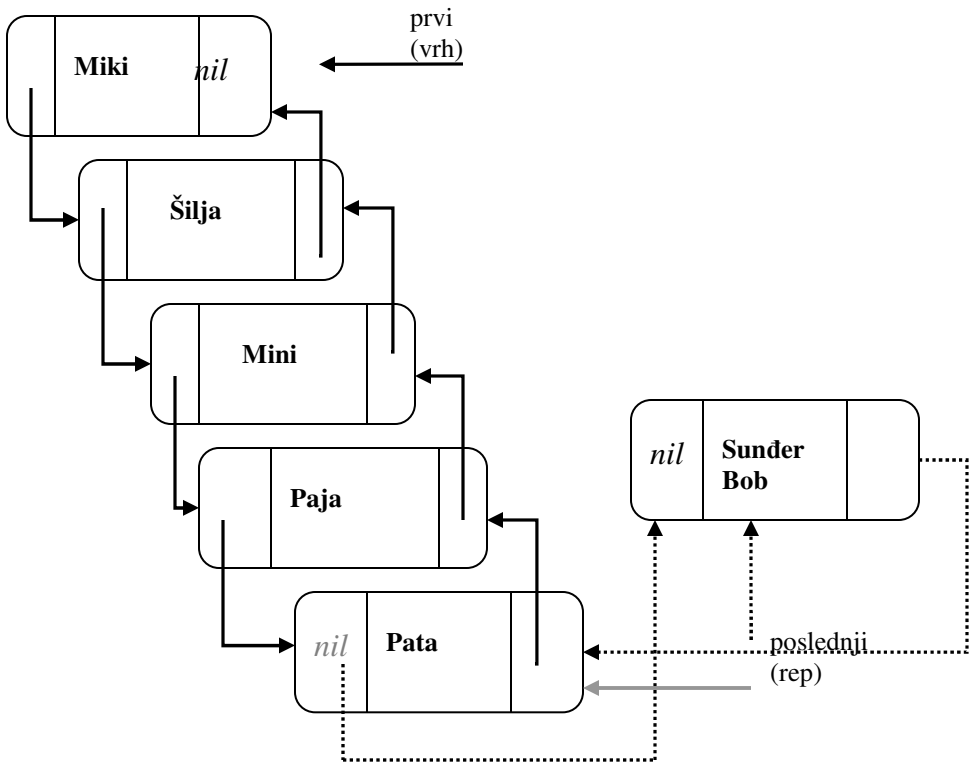


Zapis kojim bismo opisali čvor jedne FIFO liste i promenljive koje označavaju njen vrh i rep bi mogli da se opišu ovako:

poslednji ↑*elementFIFO*
prvi ↑*elementFIFO*
zapis *elementFIFO*
    reč **Niska,**
    prethodni ↑*elementFIFO*
    sledeći ↑*elementFIFO*
kraj zapisa

S obzirom da se FIFO listi može pristupiti samo na jedan određeni način, preko vrha liste za čitanje i uklanjanje elmenata, odnosno preko repa za dodavanje novog, za rad sa njom su potrebne samo dve operacije: dodavanje čvora na listu i skidanje čvora s liste.

Da bi se dodao element na rep FIFO liste, postupa se na poptuno isti način kao i kod LIFO liste, samo što treba podesiti i pokazivač sledećeg elementa: raniji rep FIFO liste pokazuje kao sledeći novododati element. Na sledećoj slici je prikazano kako bi izgledala naša FIFO lista kada bi u banku posle Mikija, Šilje, Mini, Paje i Pate stigao i Sunđer Bob koji staje na kraj reda za šalter.

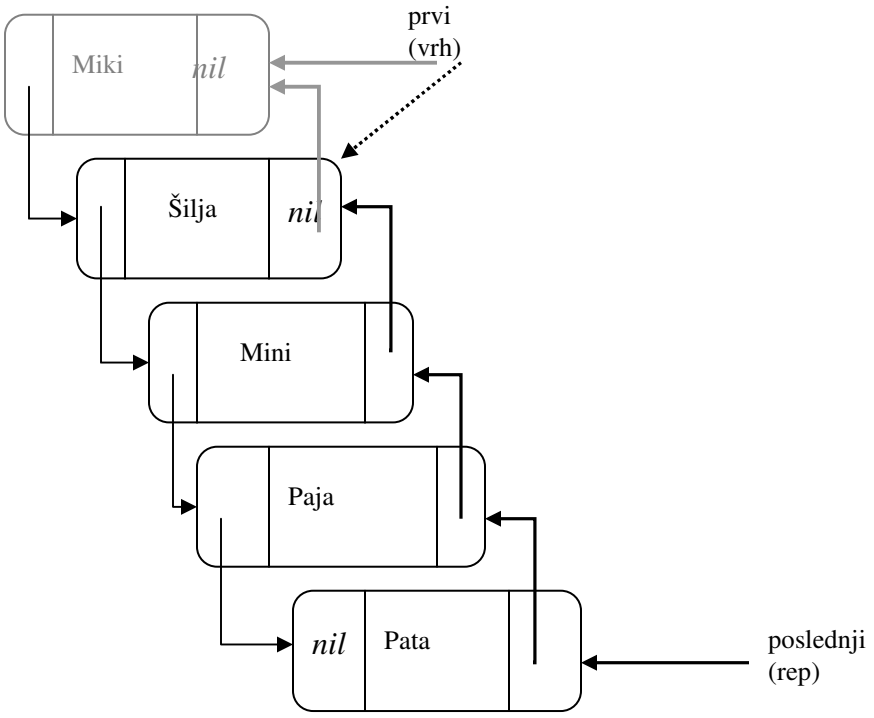


„Dodaj element *x* na rep FIFO liste“
poslednji, prvi i novi su pokazivači elementa liste - ↑*elementFIFO*
x je niska (**Niska**)
noviElementListe(*novi*)
novi↑*.reč* ← *x*
novi↑*.prethodni* ← *poslednji*
novi↑*.sledeći* ← nil
„Treba još da se proveri da li je lista pre dodavanja novog bila prazna“
Ako *prvi* = nil **onda**
    prvi ← *novi*
inače
    „Stari poslednji kao sledećeg pokazuje *novi*“
    poslednji↑*.sledeći* ← *novi*
kraj ako-onda
poslednji ← *novi*

☺

Da bi se uklonio element sa vrha FIFO liste, treba da se pročita sadržaj čvora koji je na vrhu liste, a to je onaj element koga pokazuje promenljiva *prvi*, a zatim treba promeniti i sadržaj promenljive *prvi* koja

sada treba da pokazuje element koga je do tada pokazivao element s vrha liste koga sada skidamo – sledeći element. Na sledećoj slici je prikazano kako bi izgledala naša FIFO lista kada bi se šalter oslobodio, Miki izašao iz reda i stao na šalter.



„Skidaj element *x* sa vrha FIFO liste“
poslednji, prvi je pokazivač elementa liste - ↑*elementFIFO*
x je niska (**Niska**)
x ← *prvi*↑*.reč*
„Treba proveriti da li posle skidanja čvora, lista ostaje prazna
ako *poslednji* = *prvi* **onda**
    prvi ← nil
    poslednji ← nil
inače
    prvi ← *prvi*↑*.sledeći*
    prvi↑*.prethodni* ← nil
kraj ako-onda
☺

Drvo

Pod *drvetom* ili *stablom* σ se podrazumeva par (**A**, **R**), gde je **A** skup nekih elemenata (ili čvorova) tipa **T**, a **R** je binarna relacija „biti otac (neposredni predak) od“ koja ima sledeće osobine:

- Među elementima skupa **A** postoji jedan, posebno istaknut element, obeležimo ga sa **K**, koji se naziva koren drveta.
- Za svaki drugi element $x \in \mathbf{A}$, postoji tačno jedna sekvencija elemenata $y_1, y_2,..., y_n$ takva da je

$$\mathbf{K} = y_1 \ \mathbf{R} \ y_2 \text{ i } y_2 \ \mathbf{R} \ y_3 \text{ i } ... \text{ i } y_{n-1} \ \mathbf{R} \ y_n = x$$

Neposredne posledice su da:

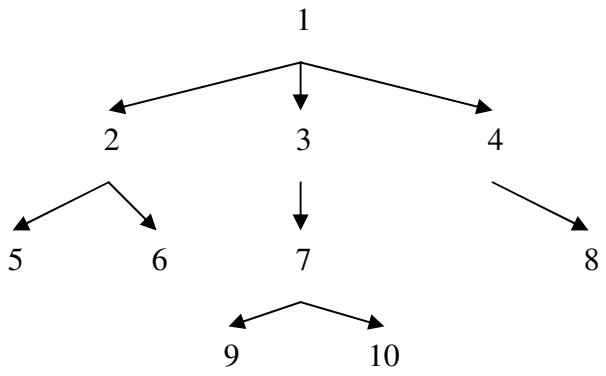
(a) Koren K nema pretke;

(b) Svako $x \in \mathbf{A} - \{\mathbf{K}\}$ ima tačno jednog neposrednog pretka.

Primer: Neka je $A = \{ x \mid 1 \leq x \leq 10\}$ i R = {(1,2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 7), (4, 8), (7, 9), (7, 10)}. Lako se proverava da je koren ovog drveta element 1 – taj element se ne nalazi kao drugi ni u jednom uređenom paru iz **R**. Takođe, za svaki drugi element *x* postoji tačno jedan put od korena do tog elementa, na primer:

$$\text{za } x = 10, \mathbf{K} = 1 \text{ i } 1 \ \mathbf{R} \ 3 \text{ i } 3 \ \mathbf{R} \ 7 \text{ i } 7 \ \mathbf{R} \ 10 \text{ i } 10 = x$$

Ovo drvo se može grafički prikazati na sledeći način:



Terminologija kojom se opisuje struktura drveta je delom pozajmljena iz genealogije:

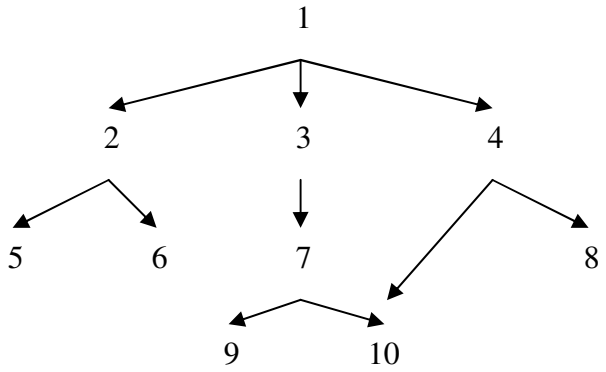
Listovi = čvorovi bez potomaka (u gronjem drvetu, to su: 5, 6, 8, 9, 10);
unutrašnji čvorovi = čvorovi koji nisu listovi (gore: 1, 2, 3, 4, 7);
grana = čvorovi na putu od korena do lista (npr, jedna grana gornjeg drveta je: 1 – 3 – 7 – 9).

Svi oni čvorovi sa putem jednake dužine od korena čine *nivo* (npr. čvorovi 5, 6, 7 i 8 su na niovu 2), a najviši nivo određuje *visinu* drveta (visina drveta iz primera je 3, jer je najduži put 1 – 3 – 7 – 9, odnosno 1 – 3 – 7 – 10). Čvorovi 2, 3, 4 su *deca* (*potomci*) čvora 1, a čvorovi 2 i 3 su *braća* čvora 4. Čvorovi 1, 3, 7 su *preci* čvora 9, a čvorovi 7, 9 i 10 su *potomci* čvora 3. *Poddrvo* čine jedan čvor drveta i svi njegovi potomci. Na primer, poddrvo sa korenom 3 bi se sastojalo od čvorova {3, 7, 9, 10} i relacije R = {(3, 7), (7, 9), (7, 10)}.

Primer: Neka je $A = \{ x \mid 1 \leq x \leq 10\}$ i R = {(1,2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 7), (4, 8), (4,10), (7, 9), (7, 10)}. Da li ova relacija R definiše drvo? Uočavamo da čvor 10 ima dva neposredna pretka: 4 i 7. To znači da se polazeći od korena 1 (koji nema pretke) do čvora 10 može doći preko dva puta:

za $x = 10$, **K** = 1 1 **R** 3 i 3 **R** 7 i 7 **R** 10
1 **R** 4 i 4 **R** 10

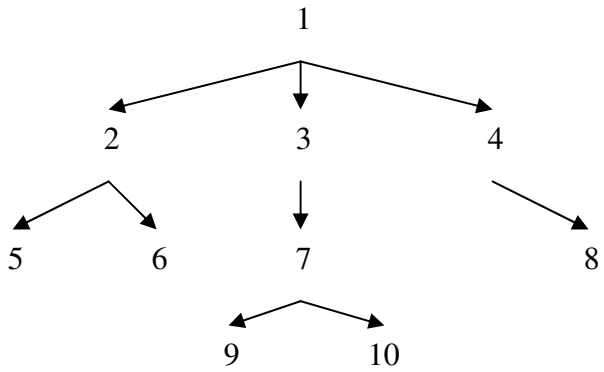
A to znači da relacija R ne definiše drvo. Ova struktura se može grafički prikazati na sledeći način:



Postavlja se pitanje kako se polazeći od korena drveta mogu posetiti svi njegovi čvorovi. Da bi se obezbedio obilazak svih čvorova postupa se na sledeći način:

- Uvek se bira leva grana drveta, dok god postoji;
- Kada ne postoji leva grana drveta, vrši se povratak na prvog pretka i bira prva naredna grana;
- Kada ne postoji naredna grana, vrši se povratak na prvog pretka, ako postoji.

Koristeći se ovim postupkom, redosled kojim bi se posetili svi čvorovi drveta iz našeg primera bio bi:



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

Iz ovog primera se vidi da su svi čvorovi drveta zaista posećeni, ali svi unutrašnji čvorovi su posećeni više puta. Postavlja se pitanje kada treba da se „pročita“ informacija koje je u tom čvoru zapisana, što je često smisao obilaženja drveta. Za to postoji više načina:

- *Prefiksni obilazak*: informacija se čita kada se na nju naiđe **prvi** put. U našem primeru redosled prefiksnog čitanja informacija bio bi:

1 2 5 2 **6** 2 1 **3 7 9** 7 **10** 7 3 1 **4 8** 4 1, ili
1, 2, 5, 6, 3, 7, 9, 10, 4, 8

- *Infiksni obilazak*: informacija se čita kada se na nju naiđe **drugi** put. Kada su u pitanju listovi, oni se uvek posećuju samo jednom pa je ta jedna poseta i prva i druga i poslednja. U našem primeru redosled infiksnog čitanja informacija bio bi:

1 2 **5 2 6** 2 **1** 3 7 **9 7 10** 7 **3** 1 4 **8 4** 1, ili
5, 2, 6, 1, 9, 7, 10, 3, 8, 4

- *Sufiksni obilazak*: informacija se čita kada se na nju naiđe **poslednji** put. U našem primeru redosled sufiksnog čitanja informacija bio bi:

1 2 **5** 2 **6 2** 1 3 7 **9** 7 **10 7 3** 1 4 **8 4 1**, ili
5, 6, 2, 9, 10, 7, 3, 8, 4, 1

Primeri upotrebe drveta su mnogobrojni: uređivanje i održavanje u uređenom stanju nasumičnih podataka, pogađanje (na primer, karata) drvetima odlučivanja, izračunavanje aritmetičkih izraza.

U praksi se najčešće koriste *binarna drveta*: to su ona drveta u kojima svaki čvor ima najviše dva potomka. Svaki čvor u binarnom drvetu sastoji se od tri podatka: od sadržaja koji je tipa **T** i od dva *pokazivača* koji pokazuju levo odnosno desno poddrvo čvora. Prilikom realizacije drveta ti pokazivači zapravo sadrže adresu u memoriji na kojoj se nalaze koreni levog, odnosno desnog, poddrveta. Prema tome, ova tri podatka zajedno čine jedan *zapis*. To bismo mogli da predstavimo na sledeći način:

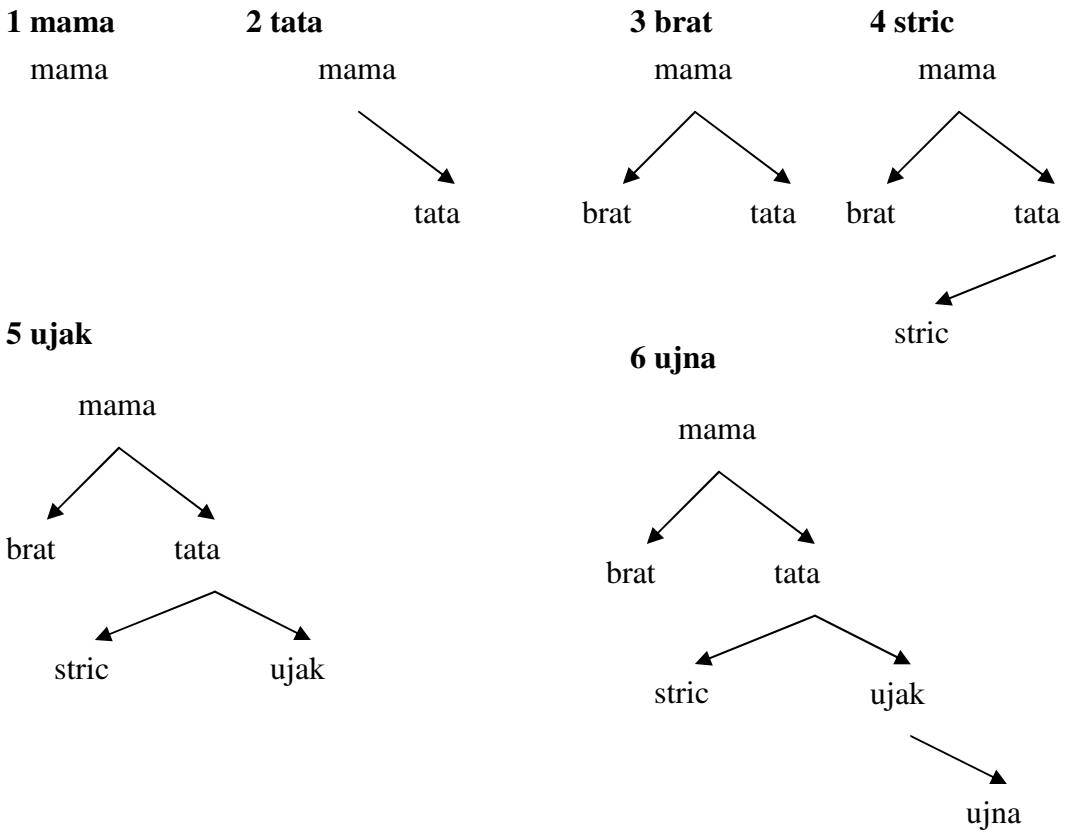
zapis *elementDrveta*
    reč **Niska,**
    levoD ↑*elementDrveta*
    desnoD ↑*elementDrveta*
kraj zapisa

Promenljiva *reč*, kao deo zapisa *elementDrveta*, je sadržaj elementa drveta (promenljiva tipa **T**), dok strelica nagore u ovom zapisu govori da je promenljiva, na primer, *levoD*, kao deo zapisa *elementDrveta*, pokazivač (tj. adresa), nekog zapisa *elementDrveta* – zapravo onog koji predstavlja koren levog poddrveta. U svim narednim primeri ćemo pretpostavljati da imamo definisan ovakav zapis.

Primer. Uređivanje podataka koji se učitavaju u nasumičnom redosledu u alfabetski redosled. Neka se sa ulaza učitavaju sledeći podaci:

mama, tata, brat, stric, ujak, ujna, sestra, strina, baka, deka

Na početku je drvo prazno, pa se prvi učitani podatak smešta u koren drveta. Sledeći učitani podatak se smešta u levi ili desni čvor, u zavisnosti od toga da li je u on u alfabetskom redosledu ispred ili iza podatka u korenom čvoru. Ako je čvor gde bi trebalo smestiti taj podatak zauzet ceo postupak se ponavlja za taj čvor. Na sledećoj slici je prikazana postupna izgradnja drveta za podatke koji se čitaju s ulaza.

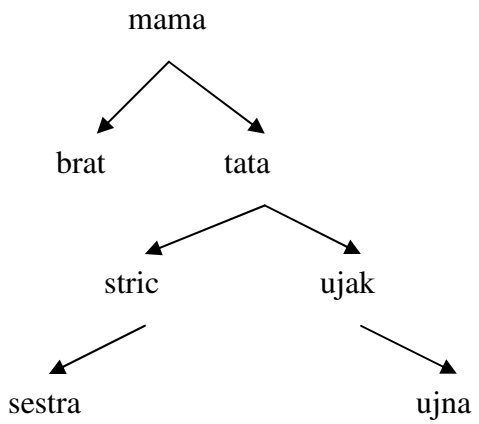


Kada se posete svi čvorovi ovog drveta i pročitaju podaci u infiksnom poretktu – kada se na podatak naiđe drugi put dobije se lista reči uređena u abecednom poretku:

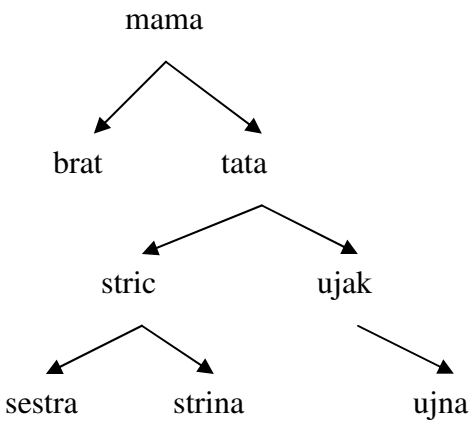
mama, brat, **baka**, **brat**, **deka**, brat, **mama**, tata, stric, **sestra**, **stric**, **strina**, stric, **tata**, ujak, **ujak**, **ujna**, tata, mama; tj.
baka, **brat**, **deka**, **mama**, **sestra**, **stric**, **strina**, **tata**, **ujak**, **ujna**

Ovaj primer pokazuje da je drvo jako pogodna struktura za uređivanje podataka koji se prikupljaju i stižu u nasumičnom poretku stalno u zadatom redosledu.

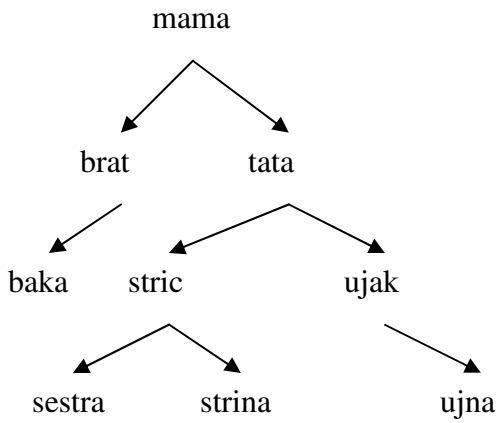
7 sestra



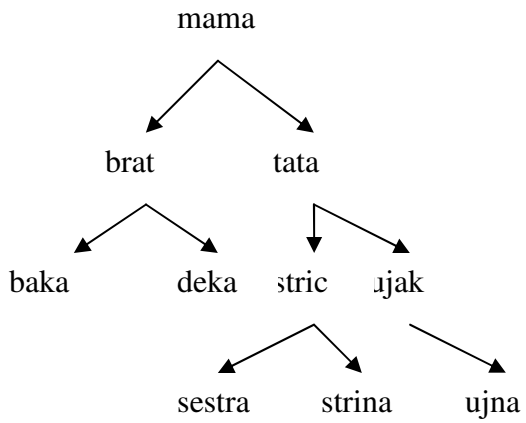
8 strina



9 baka



10 deka



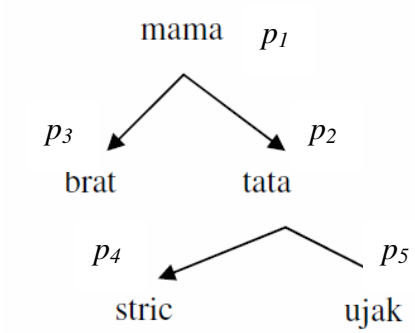
Pitanje: Ako bi kao sledeći podatak u drvo trebalo smestiti *jetrva*, gde bi se taj podatak našao?

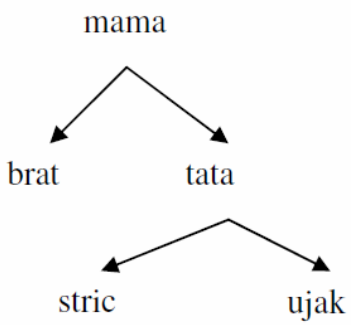
Ako pretpostavimo da nam je čvor drveta zadat zapisom *elementDrveta* koji smo ranije definisali onda bi procedura koja u infiksnom poretku ispisuje sadržaj svih čvorova drveta bila sledeća rekurzivna procedura.

„Prikaži sve elemente drveta“
Procedura *IspišiDrvo*(*koren*: ↑*elementDrveta*)
e je pokazivač elementa drveta: ↑*elementDrveta*
e ← *koren*
ako je *e* <> nil **onda**
    IspišiDrvo(*e*↑.*levo*)

prikaži *e*↑.*reč*
IspišiDrvo(*e*↑.*desno*)
kraj ako-onda
kraj procedure

Ilustrovaćemo kako bi se izvršavala ova procedura za slučaj drveta 5 sa prethodne slike. Neka su adrese čvorova p_1, p_2, p_3, p_4, p_5 onim redom kojim su čvorovi formirani i ušli u drvo, pa sadrže redom: mama, tata, brat, stric, ujak.

Pozovi *IspišiDrvo* (p_1) I poziv $e \leftarrow p_1$ pozovi *IspišiDrvo* (p_3) *levo* II-1 poziv $e \leftarrow p_3$ pozovi *IspišiDrvo* (nil) *levo* III-1(II-1) poziv $e \leftarrow$ nil povratak u II poziv prikaži p_3.*reč* **brat** pozovi *IspišiDrvo* (nil) *desno* III-2(II-1) poziv $e \leftarrow$ nil povratak u II poziv povratak i I poziv prikaži p_1.*reč* **mama** pozovi *IspišiDrvo* (p_2) *desno* II-2 poziv $e \leftarrow p_2$ pozovi *IspišiDrvo* (p_4) *levo* III-1(II-2) poziv $e \leftarrow p_4$ pozovi *IspišiDrvo* (nil) *levo* IV-1(III-1(II-2)) poziv $e \leftarrow$ nil povratak u III-1(II-2) poziv prikaži p_4.*reč* **stric** pozovi *IspišiDrvo* (nil) *desno* III-2(II-2) poziv $e \leftarrow$ nil	**5 ujak** 

povratak u III-2(II-2) poziv povratak II-2 poziv prikaži p_2.*reč* **tata** pozovi *IspišiDrvo* (p_5) *desno* III-3(II-2) poziv $e \leftarrow p_5$ pozovi *IspišiDrvo* (nil) *levo* IV-1(III-3(II-2)) poziv $e \leftarrow$ nil pov. u III-3(II-2) poziv prikaži p_5.*reč* **ujak** pozovi *IspišiDrvo* (nil) *desno* III-4(II-2) poziv $e \leftarrow$ nil povratak u III-4(II-2) poziv povratak II-2 poziv povratak I poziv povratak	**5 ujak** 

Ako bismo želeli da ispišemo sadržaj čvorova drveta u nekom drugom redosledu – sufiksnom ili prefisknom – dovoljno je da u izloženoj proceduri *IspišiDrvo* samo promene redosled iskazi za rekurzivne pozive i ispisivanje, i to za sufiksni poredak:

IspišiDrvo(*e*↑.*levo*)
IspišiDrvo(*e*↑.*desno*)
prikaži *e*↑.*reč*

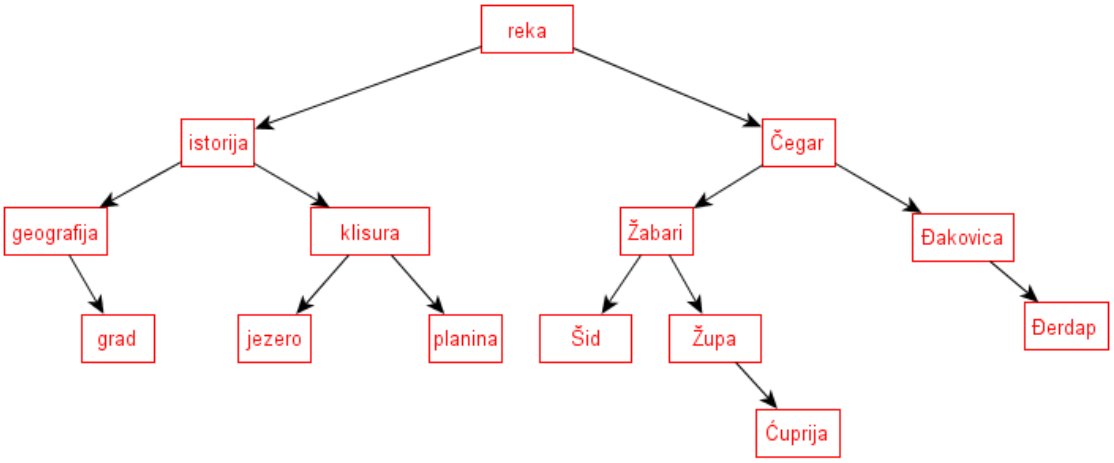
a za prefisni poredak:

prikaži *e*↑.*reč*
IspišiDrvo(*e*↑.*levo*)
IspišiDrvo(*e*↑.*desno*)

Procedura koja dodaje novi čvor u binarno pretraživačko drvo (koje je uređeno kada se obilazi u infiksnom poretku) – ako ta vrednost u drvetu već ne postoji (sama procedura ne ispituje da li vrednost postoji ili ne u drvetu) se može ovako zapisati:

```
„Dodaj vrednost x drvetu na pravo mesto“
Procedura DodajDrvetu(koren: ↑elementDrveta, x: Niska)
e, novi je pokazivač elementa drveta: ↑elementDrveta
e ← koren
ako je x < e↑.reč onda
        ako je e↑.levo = nil onda
                noviElementDrveta(novi)
                novi↑.reč ← x
                e↑.levo ← novi
        inače
                DodajDrvetu(e↑.levo, x)
        kraj ako-onda
inače
        ako je e↑.desno = nil onda
                noviElementDrveta(novi)
                novi↑.reč ← x
                e↑.desno ← novi
        inače
                DodajDrvetu(e↑.desno, x)
        kraj ako-onda
kraj ako-onda
kraj procedure
```

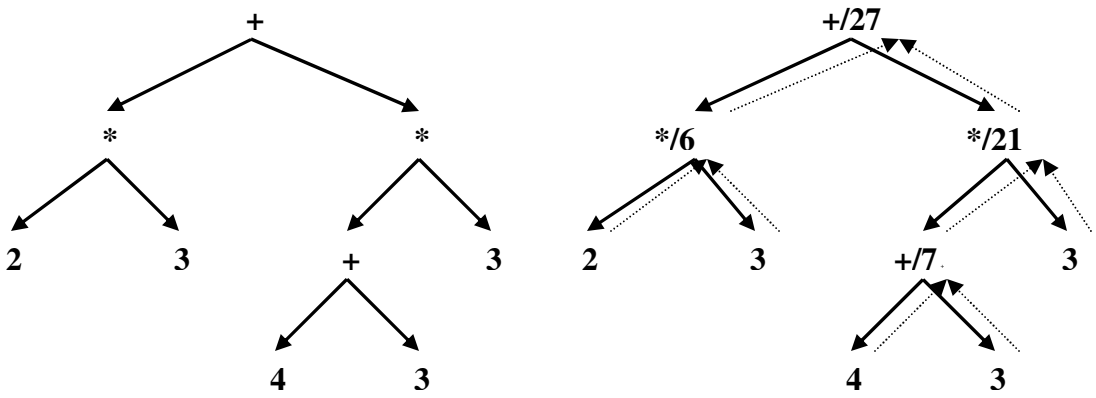
Primer. Kažemo da je niska ***a*** manja od niske ***b*** (odnosno da je niska ***b*** veća od niske ***a***) ako je ***a*** ≤ ***b***, gde je ≤ leksikografski poredak zasnovan na dopunjenoj engleskoj azbuci u kojoj važi sledeći redosled slova: A ≤ … ≤ Z ≤ a ≤ …≤ z ≤ Š ≤ Ž ≤ š ≤ ž ≤ Ć ≤ Č ≤ Đ ≤ ć ≤ č ≤ đ. Neka se na ulazu pojavljuju sledeće niske: *reka, istorija, Čegar, geografija*, *Đakovica, grad, klisura, Đerdap, Žabari*, *jezero, Župa, planina, Ćuprija*, *Šid.* Konstruisati i grafički predstaviti binarno drvo tako da je svaka niska proizvoljnog čvora drveta veća od niske ma kog levog potomka, a manja od niske ma kog desnog potomka. Svaki novi čvor se dodaje kao list do tad kreiranog drveta, poštujući red niski na ulazu. Navesti šta se dobija **infiksnim** obilaskom ovog drveta.



reka istorija geografija nil ***geografija*** *grad nil* ***grad*** *nil grad geografija* ***istorija*** *klisura jezero nil* ***jezero*** *nil jezero* ***klisura*** *planina nil* ***planina*** *nil planina klisura istorija* ***reka*** *Čegar Žabari Šid nil* ***Šid*** *nil Šid* ***Žabari*** *Župa nil* ***Župa*** *Ćuprija nil* ***Ćuprija*** *nil Ćuprija Župa Žabari* ***Čegar*** *Đakovica nil* ***Đakovica*** *Đerdap nil* ***Đerdap*** *nil Đedrap Đakovica Čegar reka*
geografija grad istorija jezero klisura planina reka Žabari Župa Ćuprija Čegar Đakovica Đerdap

Primer. Kažemo da je niska ***a*** manja od niske ***b*** (odnosno da je niska ***b*** veća od niske ***a***) ako je ***a*** ≤ ***b***, gde je ≤ leksikografski poredak zasnovan na srpskoj latinici (0 ≤ 1 ≤... 9 ≤ A ≤ B ≤ C ≤ Č ≤ Ć ≤ D ≤ Dž ≤ Đ ≤ E ≤... ≤ S ≤ Š ...≤ Z ≤ Ž ≤ a ≤ b ≤ c ≤ č ≤ ć ≤ d ≤ dž ≤ ž ≤ e ≤ ... ≤ s ≤ š ...≤ z ≤ ž). Neka se na ulazu pojavljuju sledeće niske: *Moris, Ševalije, Luj, Žurden, Alida, Vali, 1984, Pek, Žiži, Vitorio, de, Sika, 1900, Ričard, Čembrlen* Konstruisati i grafički predstaviti binarno drvo tako da je svaka niska proizvoljnog čvora drveta veća od niske ma kog levog potomka, a manja od niske ma kog desnog potomka. Svaki novi čvor se dodaje kao list do tad kreiranog drveta, poštujući red niski na ulazu. Navesti šta se dobija **infiksnim** obilaskom ovog drveta.

Izračunavanje aritmetičkih izraza kakav je, na primer, 2*3+(4+3)*3 može se takođe prikazati drvetom na čijim su listovima operandi, a u unutrašnjim čvorovima operatori izraza. Ako se podaci u drvetu čitaju u infiksnom poretku dobijamo izračunavanje izraza na uobičajeni način, pri čemu se kod poslednjeg obilaska unutrašnjeg čvora operator zamenjuje međurezultatom. Na slici je prikazano drvo izračunavanja za aritmetički izraz 2*3+(4+3)*3.



Za aritmetičke izraze kao što je 2*3+(4+3)*3 sa kojima se tipično susrećemo u matematici i programiranju, kažemo da su zapisani u *infiksnoj formi*, s obzirom da se operator nalazi između operanada. Ovaj zapis je vrlo pogodan za čoveka, ali njegova mana sa stanovišta automatskog izračunavanja je što se moraju uspostaviti konvencije o prioritetu pojedinih operatora, na primer, konvencija da množenje ima veći prioritet od sabiranja i, eventualno, koristiti zagrade za promenu prioriteta.

U računarstvu je od velikog značaja drugačiji zapis aritmetičkih izraza koji je poznat pod imenom *postfiksna* ili *obrnuta poljska notacija* u kome se operatori navode posle operanada. Ova notacija je naziv dobila u čast poljskog matematičara Jana Lukasiewitza (Jan Lukasjevič ili Jan Lukašijevič?) koji je zapravo predložio obrnut princip, da se operatori pišu ispred operanada. Na primer, tri jednostavna aritmetička izraza bi se na sledeći način zapisala u obrnutoj poljskoj notaciji:

2+3	2 3 +
2+3*4	2 3 4 * +
(2+3)*4	2 3 + 4 *

Treba uočiti da se obrnuta poljska notacija dobija kada se drvo aritmetičkog izraza zapisanog u infiksnoj notaciji pročita u sufiksnom poretku. Na primer, kada se u sufiksnom poretku pročita drvo aritmetičkog izraza 2*3+(4+3)*3 koje je prikazano na prethodnoj slici dobija se

```
2 3 * 4 3 + 3 * +
```

što i jeste obrnuta poljska notacija tog izraza.

Izračunavanje izraza u obrnutoj poljskoj notaciji izvodi se tako da se svaki operand na koji se naiđe dodaje na kraj LIFO liste koja je na početku prazna. Prilikom nailaska na binarni operator, poslednja dva elementa iz LIFO liste se brišu, operacija se izvršava sa ta dva elementa i na vrh LIFO liste se upisuje rezultat. Broj koji na kraju izračunavanja ostane u LIFO listi je rezultat izračunavanja – šta više, taj broj mora da bude jedini podatak u listi ako je izraz ispravno zapisan. U isto vreme se aritmetički izraz zapisan u obrnutoj poljskoj notaciji tretira kao FIFO lista.

Izračunavanje našeg izraza u obrnutoj poljskoj notaciji teklo bi ovako:

FIFO lista izraza	**LIFO lista izračunavanja**
2 3 * 4 3 + 3 * +	ε
3 * 4 3 + 3 * +	2
* 4 3 + 3 * +	3 2
4 3 + 3 * +	* 3 2 => 6
3 + 3 * +	4 6
+ 3 * +	3 4 6
3 * +	+ 3 4 6 => 7 6
* +	3 7 6
+	* 3 7 6 => 21 6
ε	+ 21 6 => **27**

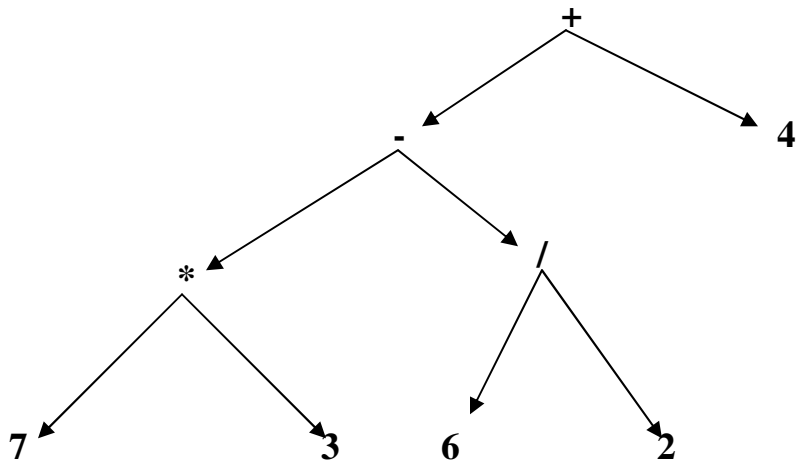
Ako je FIFO lista formirana, računanje aritmetičkog izraza preko LIFO liste se može predstaviti sledećim algoritmom u pseudo-kodu koji koristi prethodno opisane fukncije za rad sa FIFO i LIFO listama.

sve dok je *prviFIFO* <> *nil* **ponavljaj**
    x = *SkidajElementFIFO*()
    ako je *x broj* **onda**
        DodajElementLIFO(*x*)
    inače *x je operator* **onda**
        y = *SkiniElementLIFO*()
        z = *SkiniElementLIFO*()
        w = *Izvrši operaciju*(*z*, *y*, *x*)
        DodajElementLIFO(*w*)
    inače „Pogrešan podatak u FIFO listi“
do ovde
rezultat = *SkiniElementLIFO*()
ako je *poslednjiLIFO* <> nil
    „Aritmetički izraz u FIFO nije dobro formiran“

do ovde

☺

Primer. Nacrtati drvo izračunavanja za aritmetički izraz 7*3 – 6/2 + 4= 21-3+4=22. Kako izgleda taj izraz u obrnutoj poljskoj notaciji (čitanje drveta u sufiksnom poretku), a kako izračunavanje koristeći FIFO i LIFO liste.

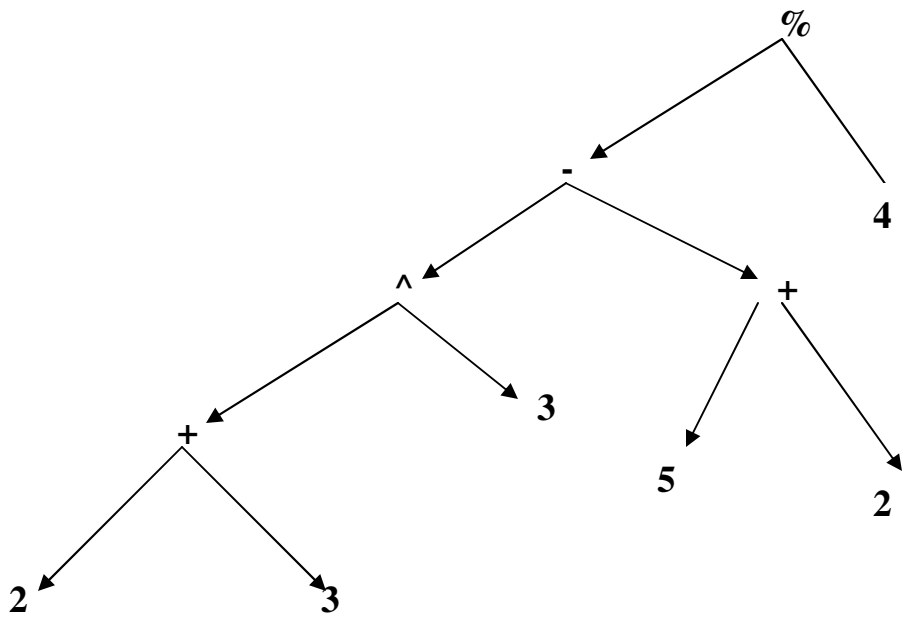


Obrnuta poljska notacija: **7 3 * 6 2 / - 4 +**

FIFO lista izraza	**LIFO lista izračunavanja**
7 3 * 6 2 / – 4 +	ε
3 * 6 2 / – 4 +	7
* 6 2 / – 4 +	3 7
6 2 / – 4 +	* 3 7 => 21
2 / – 4 +	6 21
/ – 4 +	2 6 21
– 4 +	/ 2 6 21 => 3 21
4 +	– 3 21 => 18
+	4 18
ε	+ 4 18 => 22

☺

Primer. Nacrtati drvo izračunavanja za aritmetički izraz ((2+3)^3 – (5+2))%4. Kako izgleda taj izraz u obrnutoj poljskoj notaciji (čitanje drveta u sufiksnom poretku), a kako izračunavanje koristeći FIFO i LIFO liste.



Obrnuta poljska notacija: **2 3 + 3 ^ 5 2 + - 4 %**

FIFO lista izraza	**LIFO lista izračunavanja**
2 3 + 3 ^ 5 2 + – 4 %	ε
3 + 3 ^ 5 2 + – 4 %	2
+ 3 ^ 5 2 + – 4 %	3 2
3 ^ 5 2 + – 4 %	+ 3 2 => 5
^ 5 2 + – 4 %	3 5
5 2 + – 4 %	^ 3 5 => 125
2 + – 4 %	5 125
+ – 4 %	2 5 125
– 4 %	+ 2 5 125 => 7 125
4 %	– 7 125 => 118
%	4 118
ε	% 118 4 => 2
	2

☺