4

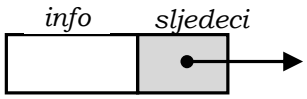
Povezane liste

U prethodnom poglavlju je uveden koncep liste, pri čemu su definirane neke tipične operacije u radu sa listama. U ovom poglavlju se opisuju **povezane liste** (ili **ulančane liste**), koje se implementiraju kao **dinamičke strukture podataka**. Povezane liste se mogu interpretirati na različite načine. U ovom poglavlju je prikazana jedna od mogućih interpretacija, koja ima za cilj da se prezentiraju neka od ključnih svojstava povezanih listi.

4.1. Jednostruko povezane liste

Povezane liste su dinamičke strukture podataka kod kojih se prostor za elemente alocira tek kada je to potrebno prilikom operacije umetanja novog elementa u listu. Nadalje, prilikom operacije brisanja, rezervirani prostor se treba osloboditi za neke druge potrebe. Kod sekvencijalne implementacije linearne liste, susjedni elementi u logičkom poretku su ujedno i susjedni elementi u fizičkom poretku. S druge strane, kod ulančane implementacije, susjedni elementi u logičkom poretku mogu biti pohranjeni bilo gdje u memoriji. Prema tome, kod ulančane implementacije linearnih listi, logički poredak elemenata se ne može izvesti iz njihovog fizičkog poretka u memoriji. Zbog toga ćemo uvesti pojam čvora, koji će sadržavati sljedeće dvije komponente (slika 4.1):

- informacioni sadržaj (označavat ćemo ga sa *info*);
- pokazivač na sljedeći čvor u logičkom poretku (označavat ćemo ga sa *sljedeci*).



Slika 4.1. Grafički prikaz čvora

Komponenta *info* se koristi za smještanje informacionog sadržaja, dok se komponenta *sljedeci* koristi za povezivanje čvorova koji sačinjavaju povezanu listu. Dakako, sadržaj komponente *info* ovisi o konkretnoj primjeni, te može zauzimati manji ili veći memorijski prostor. Radi jasnoće prikaza, kao i fokusiranja na suštinske elemente pri opisu pojedinih

algoritama, u ovoj knjizi će ilustracije u pravilu koristiti cjelobrojne podatke, bez gubitka na općenitosti.

Nadalje, na ovom mjestu ćemo uvesti funkciju GETNODE, kojom se alocira memorijski prostor za jedan čvor i vraća adresa tog prostora (pokazivač na čvor). Na primjer, sljedećom naredbom u pseudojeziku:

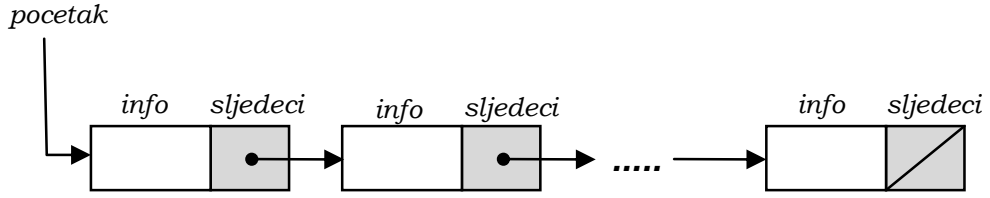
p = GETNODE (),

alocira se memorijski prostor za čvor, te se vraća adresa, koja se pridružuje pokazivaču p. Nadalje, za oslobađanje alociranog prostora na koji pokazuje pokazivač p, u pseudokodu ćemo koristiti funkciju FREENODE. Na primjer, naredbom:

FREENODE (p),

oslobađa se memorijski prostor na koji pokazuje pokazivač p.

Povezanoj listi se pristupa preko pokazivača *pocetak*, koji pokazuje na prvi čvor. Ovaj pokazivač se često naziva i čelo (eng. *head*). Ako je povezana lista prazna, onda pokazivač *pocetak* ne pokazuje niti na jedan čvor (*pocetak*=NIL). Zadnji čvor u listi, u komponenti *sljedeci*, sadrži prazan pokazivač, što ujedno označava kraj liste. Lista koja je prikazana na slici 4.2 se naziva **jednostruko povezana lista**, jer svaki čvor liste pokazuje samo na svog sljedbenika.



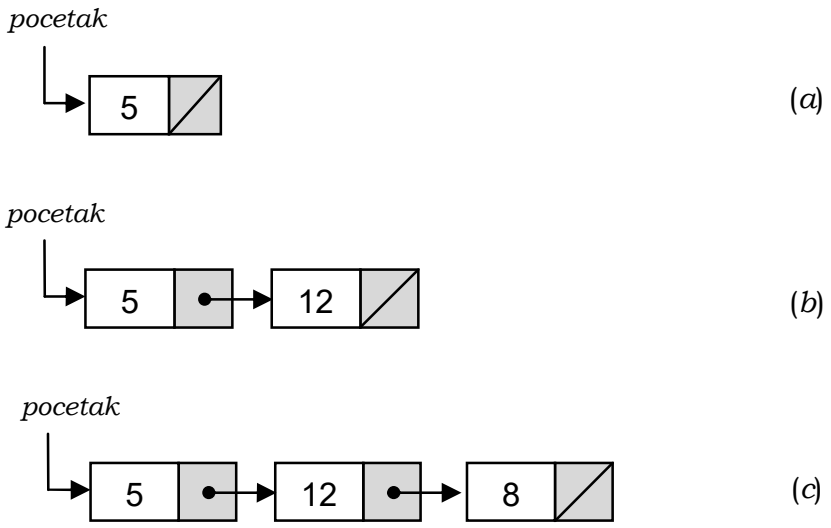
Slika 4.2. Jednostruko povezana lista

Kreirajmo sada povezanu listu (označimo je sa L), koja sadrži tri čvora i koja je prikazana na slici 4.3*c*. Jedan od načina da se kreira ova lista je da prvo kreiramo čvor koji sadrži broj 5, zatim čvor koji sadrži broj 12 i na kraju čvor koji sadrži broj 8. Radi uvođenja čitaoca u značenje nekih naredbi iz pseudojezika koji je korišten u ovoj knjizi, niz naredbi, kojima se kreiraju i povezuju tri prethodno navedena čvora, će biti prikazan upravo u pseudokodu.

Sljedećom sekvencom naredbi ćemo kreirati prvi čvor liste L i upisati sadržaj u taj čvor:

```
1   L.pocetak = GETNODE ()
2   L.pocetak.info = 5
3   L.pocetak.sljedeci = NIL
```

Prvom naredbom smo alocirali prostor za prvi čvor i pokazivaču *pocetak* smo pridružili adresu tog prostora. Drugom naredbom smo, u informacioni dio čvora, upisali vrijednost 5, dok smo trećom naredbom u pokazivač *sljedeci* upisali vrijednost NIL (slika 4.3*a*).



Slika 4.3. Kreiranje jednostruko povezane liste

Drugi čvor u listi *L*, koji sadrži broj 12, možemo kreirati sljedećom sekvencom naredbi:

```
1  L.pocetak.sljedeci = GETNODE ()
2  L.pocetak.sljedeci.info = 12
3  L.pocetak.sljedeci.sljedeci = NIL
```

Prvom naredbom smo alocirali prostor za drugi čvor i pokazivaču, smještenom u polju *sljedeci* prvog čvora, smo pridružili adresu tog prostora, te smo na taj način povezali prvi i drugi čvor. Drugom naredbom smo, u informacioni dio čvora, upisali vrijednost 12, dok smo trećom naredbom u pokazivač *sljedeci* upisali vrijednost NIL (slika 4.3*b*). Primijetite da smo poljima *info* i *sljedeci* drugog čvora pristupili preko pokazivača *pocetak*, koji pokazuje na prvi čvor.

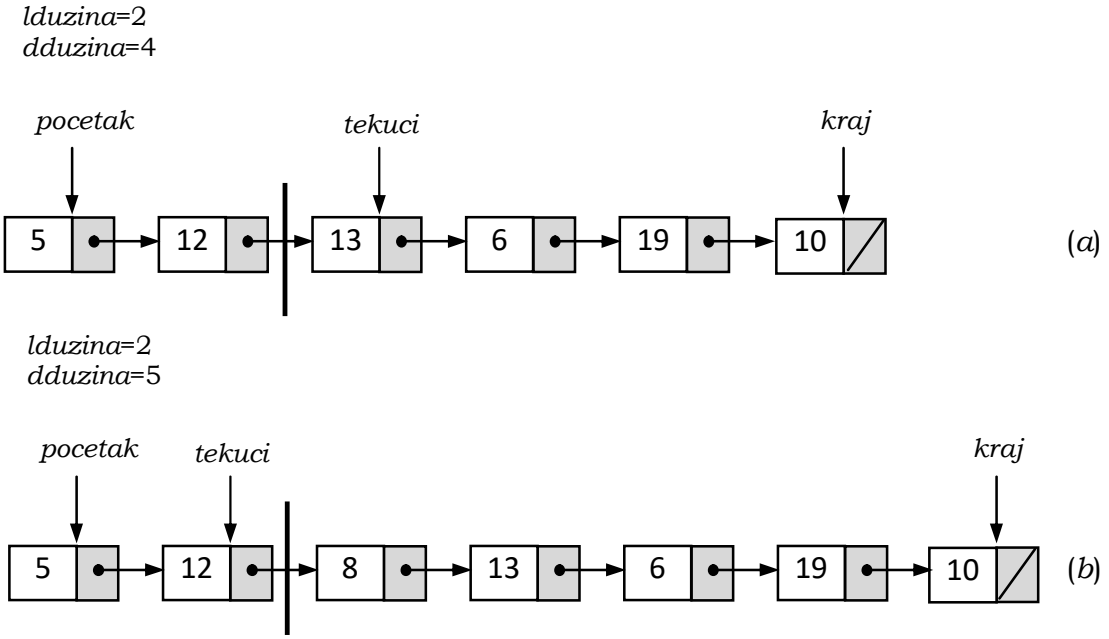
Povezanu listu *L* ćemo proširiti, dodavanjem i trećeg čvora, sljedećom sekvencom naredbi:

```
1  L.pocetak.sljedeci.sljedeci = GETNODE ()
2  L.pocetak.sljedeci.sljedeci.info = 8
3  L.pocetak.sljedeci.sljedeci.sljedeci = NIL
```

Istaknimo da smo poljima *info* i *sljedeci* trećeg čvora ponovno pristupili preko pokazivača *pocetak*, koji pokazuje na prvi čvor. Dakle, možemo zaključiti da elementima liste *L*, koja je organizirana na način da se novi čvor dodaje na kraj liste, možemo pristupiti jedino koristeći vanjski pokazivač *pocetak*. Osim toga, što je lista duža, potrebno je koristiti sve više pokazivača zapisanih u polju *sljedeci,* radi pristupa elementima liste.

Jedan od načina, da se efikasnije pristupa krajevima liste, je korištenje dva pokazivača: jedan koji će pokazivati na čelo liste (*pocetak*) i jedan koji će pokazivati na začelje liste (*kraj*). Na slici 4.4*a* je prikazan primjer jednostruko povezane liste, koja sadrži šest elemenata. Osim toga, slično kao što smo kod sekvencijalne liste uveli atribut *tekuci*, koji predstavlja granicu koja dijeli listu na dvije particije, tako ćemo se i kod povezane liste

odlučiti za korištenje tog atributa. Međutim, postavlja se pitanje kako prikazati tu granicu. Razmotrimo situaciju da za prikaz granice koristimo pokazivač koji pokazuje na prvi element desne particije, te da u listu umećemo element 8. Da bismo umetnuli čvor 8, potrebno je u komponenti *sljedeci* čvora 12 (prethodnik čvora na koji pokazuje *tekuci*) promijeniti pokazivač, tako da pokazuje na novi čvor. Budući da se pristup prethodniku nekog čvora može ostvariti tako da se slijedi lanac pokazivača počevši od čela liste, zaključujemo da pristup prethodniku nije efikasan. Međutim, ako se odlučimo da pokazivač *tekuci* pokazuje na zadnji element lijeve particije, kao što je to prikazano na slici 4.4*b*, tada nećemo imati poteškoće pri umetanju novog elementa na početak desne particije. Dodatno ćemo još, radi praćenja broja prisutnih elemenata u lijevoj i desnoj particiji, koristiti dvije cjelobrojne vrijednosti: *lduzina* (za lijevu particiju) i *dduzina* (za desnu particiju). Naime, ove vrijednosti trebamo čuvati eksplicitno i ažurirati ih u skladu sa operacijama koje modificiraju listu. Dakako, broj prisutnih elemenata u listi možemo utvrditi i slijeđenjem lanca pokazivača od čela liste, pa do njenog kraja, što bi bilo svakako neefikasno.



Slika 4.4. Ilustracija loše implementirane jednostruko povezane liste u kojoj pokazivač *tekuci* pokazuje na prvi čvor desne particije. U listu su uključeni dodatni atributi: *kraj*, *tekuci*, *lduzina* i *dduzina*. (*a*) Povezana lista prije umetanja elementa 8. (*b*) Stanje liste nakon umetanja elementa 8

4.2 Operacije sa jednostruko povezanim listama

Tipične operacije u radu sa povezanim listama su:

- pristup čvorovima u listi;
- umetanje novog čvora na tekuću poziciju;
- dodavanje novog čvora na kraj liste;
- brisanje čvora na tekućoj poziciji;
- pretraživanje liste.

4.2.1 Pristup čvorovima u listi

Operacija pristupa sljedećem čvoru je opisana procedurom IDI-NA-SLJEDECI u tabeli 4.1. Ako je desna particija prazna, operacija ne proizvodi nikakav učinak. U suprotnom, pokazivač *tekuci* se pomjera na sljedeći čvor u listi (linija 5), te se ažuriraju (linije 7-8) atributi *lduzina* (povećava se za 1) i *dduzina* (smanjuje se za 1). Procedura u obzir uzima poseban slučaj, koji se pojavljuje kada je lijeva particija prazna, u kojem se adresa za *tekuci* dobija tako da se direktno postavlja na vrijednost pokazivača *pocetak* (linija 3). Potrebno vrijeme za izvođenje ove operacije je reda $O(1)$.

Tabela 4.1

```
    IDI-NA-SLJEDECI (L)
1   if (L.dduzina ≠ 0) then
2      if (L.lduzina == 0) then
3         L.tekuci = L.pocetak
4      else
5         L.tekuci = L.tekuci.sljedeci
6      end_if
7      L.lduzina = L.lduzina + 1
8      L.dduzina = L.dduzina - 1
9   end_if
```

Operacija pristupa prethodnom čvoru je opisana procedurom IDI-NA-PRETHODNI u tabeli 4.2. Budući da kod jednostruko povezane liste ne možemo direktno pristupiti prethodnom čvoru, operacija pristupa prethodniku se ostvaruje slijeđenjem lanca pokazivača, počevši od početka liste (linija 5), pa sve do prethodnika razdvojnog čvora. Na kraju se ažuriraju (linije 11-12) atributi *lduzina* (smanjuje se za 1) i *dduzina* (povećava se za 1). Operacija uzima u obzir i poseban slučaj, koji se pojavljuje kada lijeva particija sadrži samo jedan element (linija 2). Tada se pokazivač *tekuci* direktno postavlja na vrijednost *tekuci* = NIL (linija 3). Ako je lijeva particija prazna, operacija ne proizvodi nikakav učinak.

Tabela 4.2

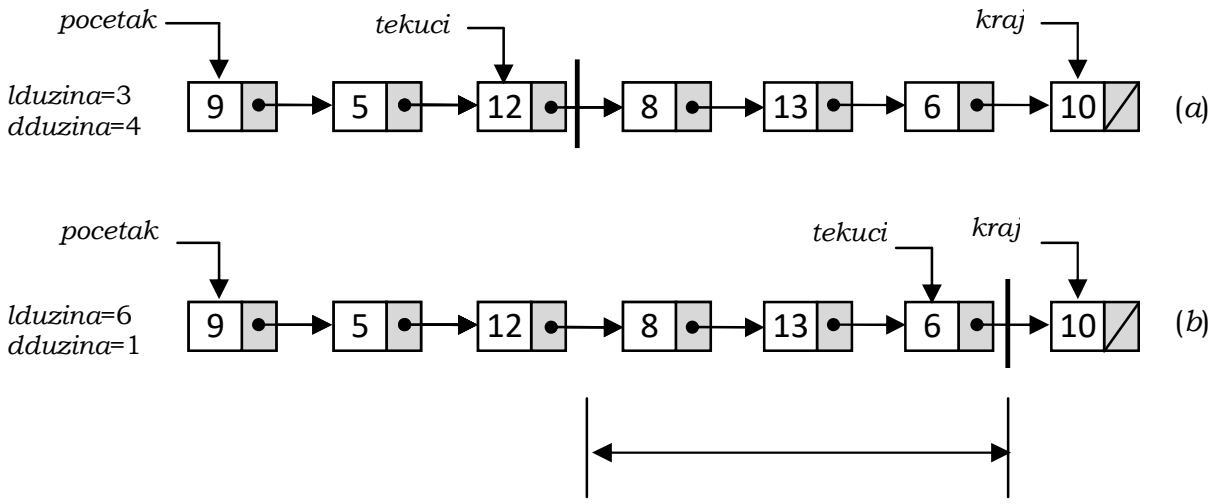
```
    IDI-NA-PRETHODNI (L)
1   if (L.lduzina ≠ 0) then
2      if (L.lduzina == 1) then
3         L.tekuci = NIL
4      else
5         p = L.pocetak
6         while (p.sljedeci ≠ L.tekuci) do
7            p = p.sljedeci
8         end_while
9         L.tekuci = p
10     end_if
11     L.lduzina = L.lduzina - 1
12     L.dduzina = L.dduzina + 1
13  end_if
```

Operacija, kojom se pokazivač *tekuci* pozicionira tako da se granica između dvije particije postavlja na *i*-toj poziciji (indeks *i*=0 označava prvu poziciju), je opisana procedurom IDI-NA-POZICIJU u tabeli 4.3. Prvo se na temelju indeksa *i*, izračunavaju nove vrijednosti atributa *dduzina* i *lduzina* (linije 4 i 5), a zatim se, slijedeći lanac pokazivača od čela liste, pokazivač *tekuci* pozicionira na zadnji čvor u lijevoj particiji (linije 9-13). Procedura u obzir uzima i specijalan slučaj, koji se pojavljuje kada indeks *i* ima vrijednost *i*=0, te se pokazivač *tekuci* tada direktno postavlja na *tekuci*=NIL (linije 6-7). Potrebno vrijeme za izvođenje ove operacije je reda $O(n)$.

Tabela 4.3

```
    IDI-NA-POZICIJU (L, i)
1   if (i < 0 or i > L.dduzina + L.lduzina) then
2      ERROR („Indeks izvan dozvoljenog raspona“)
3   end_if
4   L.dduzina = L.dduzina + L.lduzina – i
5   L.lduzina = i
6   if ( i == 0 ) then
7      L.tekuci = NIL
8   else
9      L.tekuci = L.pocetak
10     for i = 0 to i-1 do
11        L.tekuci = L.tekuci.sljedeci
12     end_for
13  end_if
```

Procedura IDI-NA-POZICIJU ostvaruje pomjeranje pokazivača *tekuci* prema željenoj poziciji, tako da se slijedi lanac pokazivača od čela liste. Međutim, u nekim situacijama se elementima liste pristupa tako da se sljedeći element kojem se pristupa nalazi na nekoj poziciji iza elementa kojem se prethodno pristupilo. Tada vrijedi da je $i > L.lduzina$. Ova procedura će i u takvim slučajevima pristup *i*-tom elementu ostvariti tako da se lanac pokazivača slijedi od čela liste, bez obzira što bi bilo efikasnije lanac pokazivača slijediti od trenutne vrijednosti pokazivača *tekuci*. Na primjer, na slici 4.5*a* je prikazana konfiguracija liste u kojoj je vrijednost *lduzina*=3. Pretpostavimo da je potrebno pristupiti sedmom (*i*=6) elementu, tako da dužina lijeve particije iznosi *lduzina*=6. Procedura IDI-NA-POZICIJU će trebati šest koraka, bez obzira što se pristup sedmom elementu zapravo mogao ostvariti u tri koraka. U tabeli 4.4 je opisana modificirana varijanta pristupa u obliku procedure IDI-NA-POZICIJU-M, koja u obzir uzima prethodno opisanu mogućnost povećanja efikasnosti pristupa.



Slika 4.5. Pristup novoj poziciji iz tekuće pozicije

Tabela 4.4

```
    IDI-NA-POZICIJU-M (L, i)
1   if (i < 0 or i > L.dduzina + L.lduzina) then
2       ERROR („Indeks izvan dozvoljenog raspona“)
3   end_if
4   p = 0;
5   ld = L.lduzina
6   L.dduzina = L.dduzina + L.lduzina – i
7   L.lduzina = i
8   if  (i == 0) then
9       L.tekuci = NIL
10      return
11  end_if
12  if (i ≥ ld) then
13      if (ld == 0) then
14          L.tekuci = L.pocetak
15      else
16          p = ld-1
17      end_if
18  else
19      L.tekuci = L.pocetak
20  end_if
21  for i = p to i -1 do
22      L.tekuci = L.tekuci.sljedeci
23  end_for
```

4.2.2 Umetanje novog čvora u listu

Operacija umetanja novog elementa x na čelo desne particije je opisana procedurom UMETNI u tabeli 4.5.

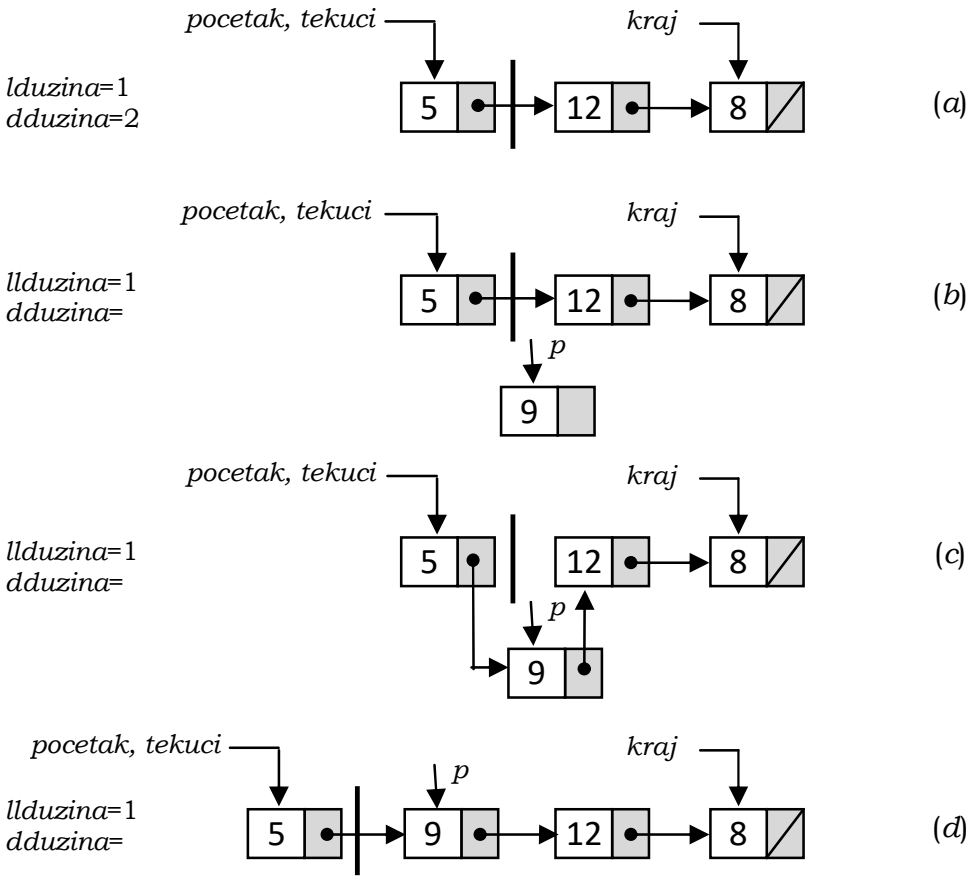
Tabela 4.5

```
    UMETNI (L, x)
1   p = GETNODE ()
2   p.info = x
3   if (L.lduzina == 0) then
4      p.sljedeci = L.pocetak
5      L.pocetak  = p
6      if (L.lduzina + L.dduzina == 0) then
7         L.kraj = p
8      end_if
9   else
10     p.sljedeci = L.tekuci.sljedeci
11     L.tekuci.sljedeci = p
12     if (L.dduzina == 0) then
13        L.kraj = L.tekuci.sljedeci
14     end_if
15  end_if
16  L.dduzina = L.dduzina + 1
```

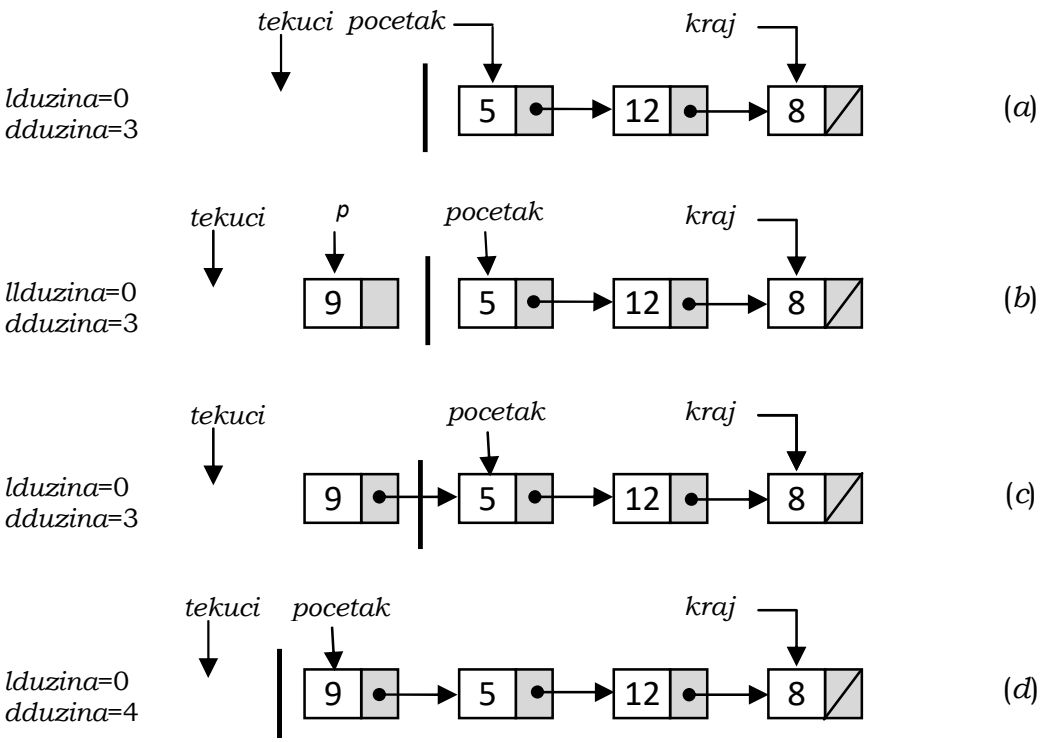
Prvo ćemo razmotriti situaciju u kojoj se element umeće na bilo koju poziciju između krajnjih čvorova. Tada operacija umetanja uključuje tri koraka. Prvo je potrebno kreirati novi čvor i u njega smjestiti novi element (linije 1-2). Nadalje, u komponentu *sljedeci* novog čvora, je potrebno upisati adresu koja upućuje na čvor koji je trenutno na čelu desne particije (linija 10), dok se u komponentu *sljedeci* zadnjeg čvora lijeve particije (na kojeg pokazuje *tekuci*), treba upisati adresa koja upućuje na novi čvor koji se umeće (linija 11). Na kraju se atribut *dduzina* ažurira tako da se povećava za 1 (linija 16).

Ovakva situacija je ilustrirana na slici 4.6 umetanjem novog čvora sa sadržajem x=9. Početno stanje liste je prikazano na slici 4.6*a*. Nakon kreiranja novog čvora i upisivanja sadržaja 9 (slika 4.6*b*), lista se prevezuje tako da novi čvor pokazuje na čvor 12, koji je prethodno bio prvi element desne particije, dok čvor 5, koji je zadnji element lijeve particije, pokazuje na novi čvor 9 (slika 4.6*c*). Na kraju je predstavljen samo drugačiji grafički prikaz prevezane liste, uz novu ažuriranu vrijednost atributa *dduzina*=3 (slika 4.6*d*).

Procedura UMETNI u obzir uzima i specijalan slučaj, koji nastaje kada se novi element umeće na početak liste, u kojem se lista treba prevezati na malo drugačiji način (linije 3-4), te u kojoj je potrebno ažurirati pokazivač *pocetak* (linija 5). Takva situacija je ilustrirana na slici 4.7, izvođenjem poziva procedure UMETNI(*L,* 9). Početno stanje je prikazano na slici 4.7*a*, dok je prvi korak (kreiranje novog čvora i upisivanje sadržaja x=9) prikazan na slici 4.7*b*. Prilikom prevezivanja liste potrebno je jedino povezati novi čvor sa čvorom 5 (slika 4.7*c*). Dodatno je potrebno ažurirati pokazivač *pocetak*, tako da pokazuje na novi čvor 9, te je na kraju potrebno povećati atribut *dduzina* za 1, jer je novi element umetnut u desnu particiju (slika 4.7*d*).

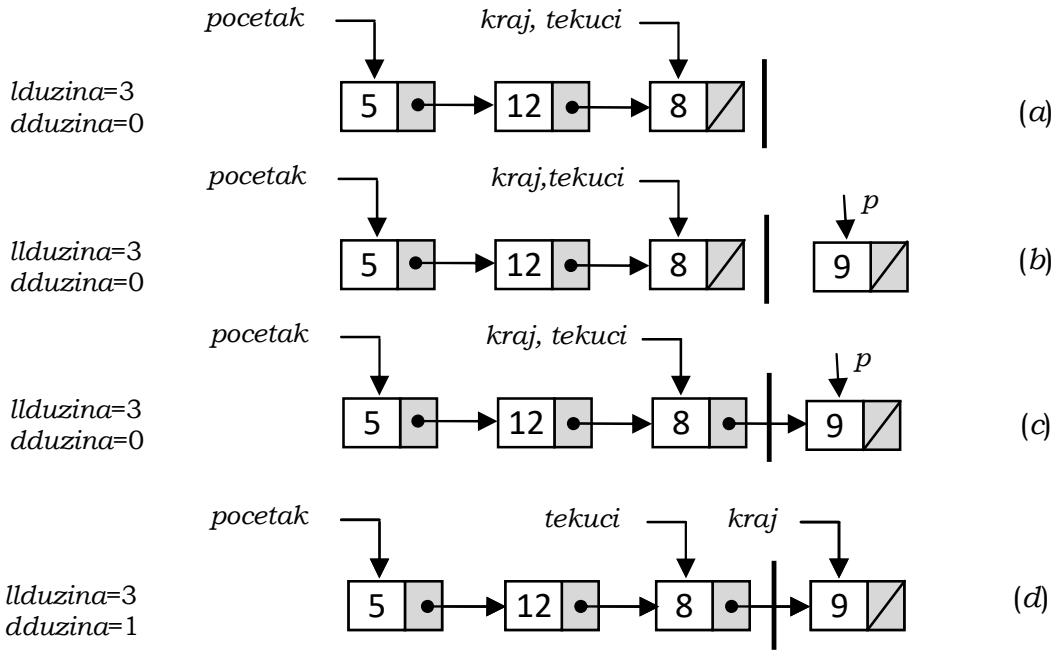


Slika 4.6. Umetanje novog čvora



Slika 4.7. Umetanje čvora na početku jednostruko povezane liste

Konačno, procedura UMETNI u obzir uzima i specijalan slučaj, koji se pojavljuje pri umetanju novog čvora na kraj liste. U tom slučaju je potrebno ažurirati pokazivač *kraj* (linije 12-13). Ovakav slučaj je ilustriran na slici 4.8. Početno stanje liste je prikazano na slici 4.8*a*. Nakon kreiranja novog čvora i upisivanja elementa 9 (slika 4.8*b*), lista se prevezuje tako da se u komponentu *sljedeci* zadnjeg čvora upisuje adresa novog čvora *p* (slika 4.8*c*). Pošto se testiranjem odgovarajućeg uvjeta utvrđuje da je desna particija prethodno bila prazna, ažurira se pokazivač *kraj*, tako da pokazuje na novi čvor, te se povećava atribut *dduzina* za 1 (slika 4.8*d*).



Slika 4.8. Umetanje novog čvora na kraj liste

4.2.3 Dodavanje novog čvora u listu

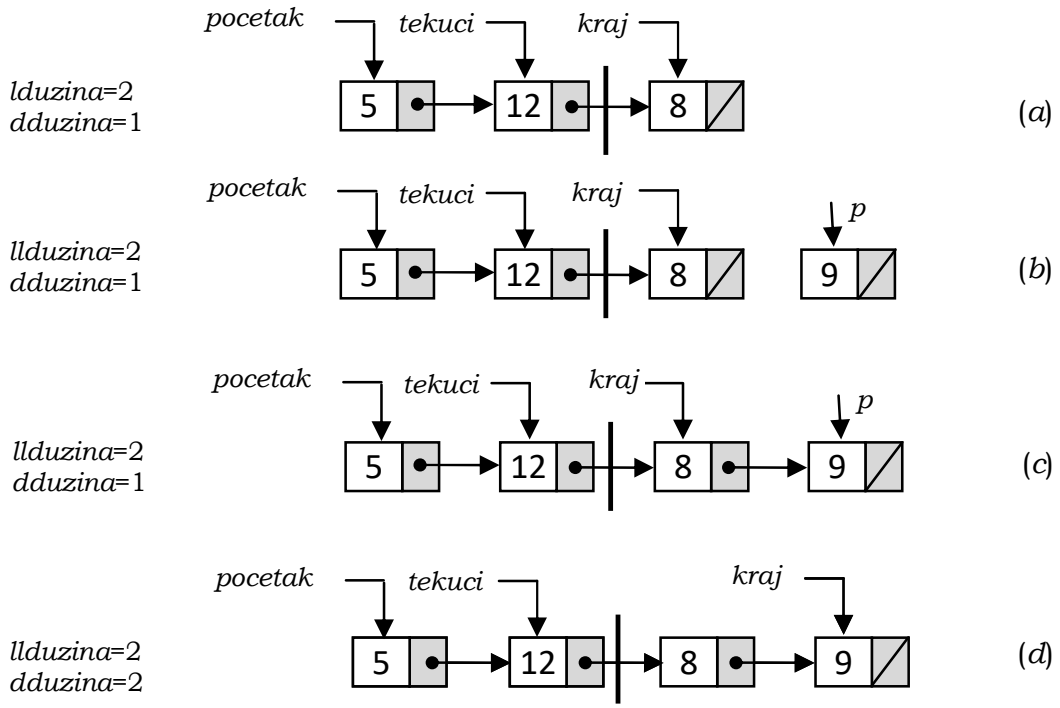
Operacija dodavanja novog elementa *x* na kraj liste je opisana procedurom DODAJ u tabeli 4.6. Prvo se kreira novi čvor i u njega smješta novi element *x*, te se u komponentu *sljedeci* upisuje NIL, jer se novi čvor dodaje na kraj (linije 1-3). Nadalje, u komponentu *sljedeci* zadnjeg čvora se upisuje adresa koja upućuje na novi čvor, a pokazivač *kraj* se ažurira tako da pokazuje na novi čvor (linija 7). Na kraju se povećava atribut *dduzina* za 1 (linija 9). Procedura DODAJ uzima u obzir i jedan specijalan slučaj, koji se pojavljuje u situaciji kada se novi čvor umeće u praznu listu, u kojem je dodatno potrebno ažurirati i pokazivač *pocetak* (linije 4-5).

Operacija dodavanja je ilustrirana na slici 4.9 izvođenjem poziva procedure DODAJ (*L*, 9), koja početnu konfiguraciju liste (5, 12 | 8) pretvara u konfiguraciju (5, 12 | 8, 9). Početno stanje je prikazano na slici 4.9*a*, dok je prvi korak (kreiranje novog čvora i upisivanje sadržaja *x*=9) prikazan na slici 4.9*b*. Prilikom prevezivanja liste potrebno je jedino povezati novi čvor sa čvorom 8 (slika 4.9*c*). Međutim, dodatno je potrebno ažurirati

pokazivač *kraj*, tako da pokazuje na novi čvor 9, te je na kraju potrebno povećati atribut *dduzina* za 1, jer je novi element umetnut u desnu particiju (slika 4.9*d*) .

Tabela 4.6

```
    DODAJ (L, x)
1   p = GETNODE ()
2   p.info = x
3   p.sljedeci = NIL
4   if (L.lduzina + L.dduzina == 0) then
5      L.pocetak = L.kraj = p
6   else
7      L.kraj = L.kraj.sljedeci = p
8   end_if
9   L.dduzina = L.dduzina + 1
```



Slika 4.9. Dodavanje novog čvora u listu

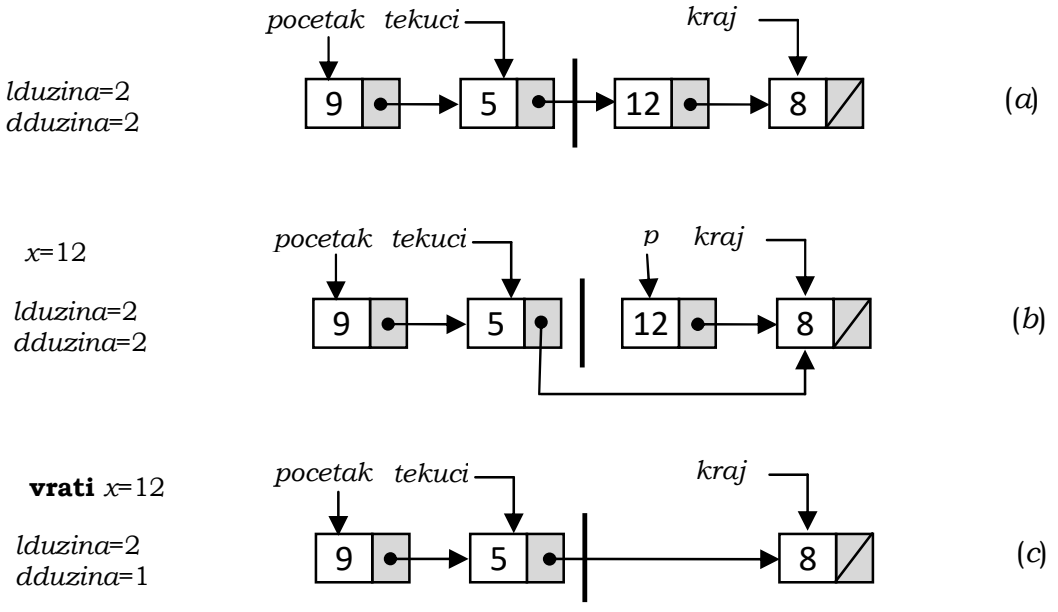
4.2.4 Izbacivanje čvora iz liste

Operacija izbacivanja uklanja prvi element iz desne particije i opisana je funkcijom IZBACI u tabeli 4.7. Funkcija vraća informacioni sadržaj čvora koji se izbacuje.

Tabela 4.7

```
    IZBACI (L)
1   if (L.dduzina == 0) then
2       ERROR(„Nema nista za izbaciti“)
3   end_if
4   if (L.lduzina == 0) then
5       x = L.pocetak.info
6       p = L.pocetak
7       L.pocetak = p.sljedeci
8   else
9       x = L.tekuci.sljedeci.info
10      p = L.tekuci.sljedeci
11      L.tekuci.sljedeci = p.sljedeci
12  end_if
13  if (L.dduzina == 1) then
14      L.kraj = L.tekuci
15  end_if
16  FREENODE (p)
17  L.dduzina = L.dduzina - 1
18  return x
```

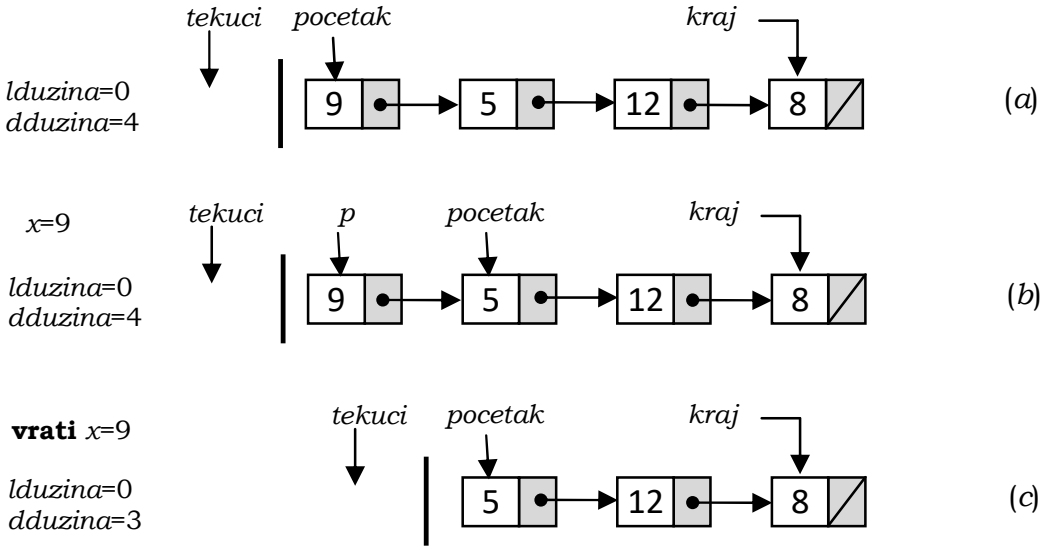
Prvo ćemo razmotriti situaciju u kojoj se izbacuje element koji se nalazi na poziciji između krajnjih čvorova. Ako je desna particija prazna, funkcija prijavljuje grešku, kojom se signalizira da se nema šta izbaciti iz liste (linije 1-3). Inače, operacija izbacivanja u privremenom objektu *x* čuva informacioni sadržaj, a lista se prevezuje tako da se adresa, iz komponente *sljedeci* čvora koji se briše, upisuje u komponentu *sljedeci* čvora na koji pokazuje *tekuci* (linija 11), pri čemu se adresa čvora koji se briše čuva u pokazivaču *p* (linija 10), radi oslobađanja memorijskog prostora koji je taj čvor zauzimao (linija 16). Na kraju se ažurira dužina desne particije, tako da se atribut *dduzina* umanji za 1 (linija 17), te se vraća informacioni sadržaj čvora koji se briše (linija 18).



Slika 4.10. Izbacivanje čvora iz jednostruko povezane liste

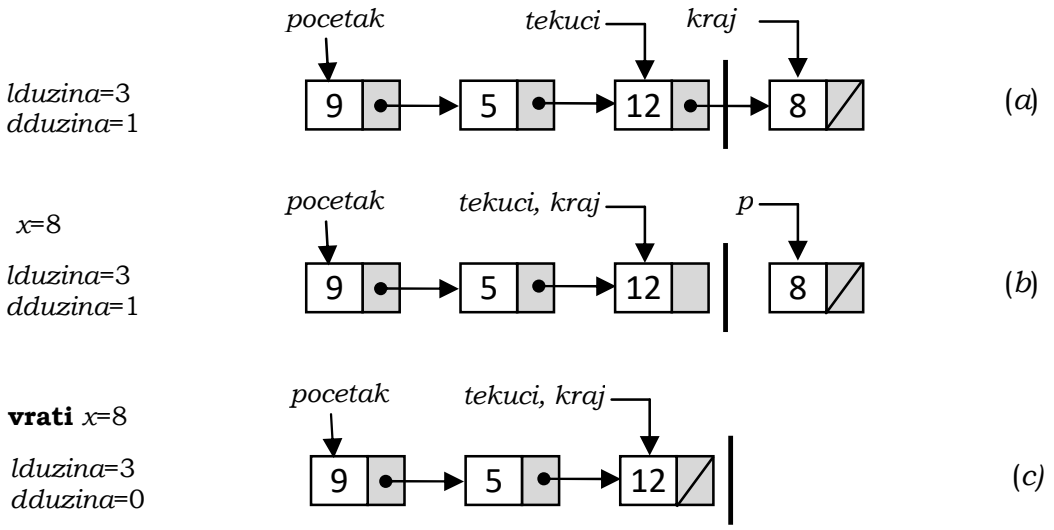
Prethodno opisana situacija je ilustrirana na slici 4.10 operacijom brisanja čvora sa sadržajem *x*=12, koja početnu konfiguraciju (9, 5 | 12, 8) pretvara u konfiguraciju (9, 5 | 8). Početno stanje liste je prikazano na slici 4.10*a*. Nakon upisivanja informacionog sadržaja 12 u privremenu varijablu *x*=12, lista se prevezuje tako da se čvor, na koji pokazuje *tekuci* (prethodnik čvora 12), povezuje sa čvorom 8, koji je bio sljedbenik čvora 12. Na taj način se čvor 12 izbacuje iz logičkog poretka liste (slika 4.10*b*). Na kraju je još i fizički uklonjen čvor 12, tako što je oslobođen memorijski prostor koji je taj čvor zauzimao, te je dužina desne particije umanjena na vrijednost *dduzina*=1 (slika 4.10*c*).

Procedura IZBACI uzima u obzir i specijalan slučaj, koji nastaje kada se izbacuje element na početku liste, u kojem nije potrebno listu prevezivati, ali je potrebno sačuvati adresu čvora koji se briše i ažurirati pokazivač *pocetak*, tako da pokazuje na sljedbenika prvog čvora (linije 4-7). Takva situacija je ilustrirana na slici 4.11 izvođenjem procedure IZBACI, koja početnu konfiguraciju liste (| 9, 5, 12, 8) pretvara u konfiguraciju (| 5, 12, 8) . Početno stanje je prikazano na slici 4.11*a*, dok je ažuriranje pokazivača *pocetak* ilustrirano na slici 4.11*b*, čime je prvi element u logičkom smislu izbačen iz liste. Na kraju se čvor 9 i fizički uklanja iz liste, te se ažurira atribut *dduzina* na vrijednost *dduzina*=3 (slika 4.11*c*).



Slika 4.11. Izbacivanje čvora na početku jednostruko povezane liste

Konačno, procedura IZBACI uzima u obzir i specijalnu situaciju, koja nastaje kada se izbacuje zadnji čvor, u kojoj je potrebno ažurirati pokazivač *kraj*, tako da pokazuje na prethodnika (čvor na koji pokazuje *tekuci*) zadnjeg čvora (linije 13-15). Ovakva situacija je ilustrirana na slici 4.12 izvođenjem procedure IZBACI, koja početnu konfiguraciju liste (9, 5, 12 | 8) pretvara u konfiguraciju (9, 15, 12 |). Početno stanje je prikazano na slici 4.12*a*, dok je ažuriranje pokazivača *kraj* ilustrirano na slici 4.12*b*, čime je zadnji element u logičkom smislu izbačen iz liste. Na kraju se čvor 8 i fizički uklanja iz liste, te se ažurira atribut *dduzina* na vrijednost *dduzina*=0 (slika 4.12*c*).



Slika 4.12. Izbacivanje čvora na kraju jednostruko povezane liste

4.2.5 Pretraživanje liste

Operacijom pretraživanja se pronalazi čvor u kojem je pohranjen neki traženi sadržaj. Ova operacija je opisana funkcijom TRAZI u tabeli 4.8. Funkcija kao argument uzima listu *L* i element *x* koji se traži. Funkcija vraća vrijednost *true*, ako je traženi element prisutan u listi, odnosno vrijednost *false*, ako traženi element nije prisutan u listi.

Tabela 4.8

```
    TRAZI (L, x)
1   if (L.lduzina + L.dduzina == 0) then
2       ERROR(„Lista je prazna“)
3   end_if
4   p = L.pocetak
5   while ((p ≠ NIL) and p.info ≠ x) do
6       p = p.sljedeci
7   end_while
8   return p ≠ NIL
```

Ako je dužina liste jednaka nuli, funkcija prijavljuje grešku kojom se signalizira da je lista prazna (linije 11-3). Funkcija TRAZI, počevši od čvora na koji pokazuje *pocetak*, linearnim pretraživanjem pronalazi prvi čvor koji sadrži element *x*. Na primjer, za liste prikazane na slici 4.4, poziv TRAZI (*L,*13) vraća *true*, dok poziv TRAZI (*L,*15) vraća *false*. Pretraživanje se obavlja sekvencijalno, počevši od čela liste (linija 4), i slijedeći lanac pokazivača sve dok se ne pronađe traženi informacioni sadržaj, ili dok se ne dođe do kraja liste (linije 5-7). Da bi se pretražila lista koja sadrži *n* čvorova, potrebno je u najgorem slučaju pretražiti čitavu listu, pa zaključujemo da je u najgorem slučaju vremenska složenost pretraživanja liste $O(n)$.

4.2.6 Obrtanje poretka čvorova u listi

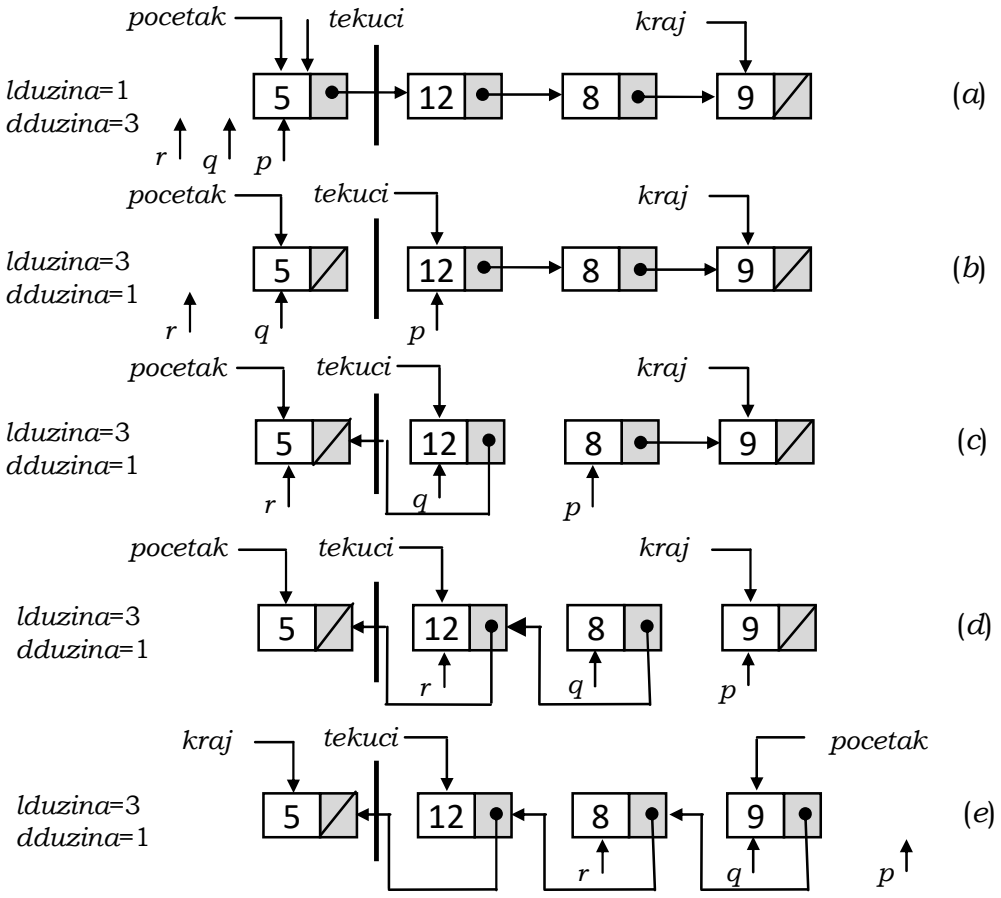
U primjeni povezanih listi se ponekada javlja potreba i za nekim dodatnim operacijama kao što su: razbijanje jedne liste u dvije liste, spajanje više listi, kopiranje listi, obrtanje poretka elemenata u listi, itd. Na ovom mjestu je upravo opisana operacija obrtanja poretka čvorova, koja invertuje poredak elemenata u listi na način da prvi čvor postaje posljednji, drugi čvor postaje predposljednji, itd. Nadalje, operaciju obrtanja ćemo definirati tako da ona podrazumijeva i „obrtanje" broja elemenata u lijevoj i desnoj particiji. Drugim riječima, atribut *dduzina*, koji predstavlja broj elemenata u desnoj particiji, će poprimiti vrijednost atributa *lduzina*, koji predstavlja broj elemenata u lijevoj particiji, i obrnuto.

Operacija obrtanja poretka čvorova u listi je opisana procedurom OBRNI-POREDAK u tabeli 4.9. Budući da nakon obrtanja zadnji element lijeve particije treba biti sljedbenik zadnjeg elementa lijeve particije prije operacije obrtanja, potrebno je ažurirati pokazivač *tekuci*, tako da pokazuje na svog sljedbenika (linija 8). Osim toga, operacija OBRNI-POREDAK uzima u obzir i specijalan slučaj, koji se pojavljuje u situaciji kada je lijeva particija prazna (linije 3-4), kao i slučaj koji se pojavljuje kada je desna particija bila prazna (linije 5-6). Nadalje, operacija koristi tri tekuća pokazivača (*r*, *q* i *p*), koji se slijedno pomjeraju od početka prema kraju liste, tako da *p* pokazuje na sljedbenika čvora na kojeg pokazuje *q*, dok *r* pokazuje na prethodnika čvora na kojeg pokazuje *q*. U svakoj iteraciji **while** petlje se pokazivač, u komponenti *sljedeci* čvora na koji pokazuje *q*, prebacuje sa sljedbenika *p* na prethodnika *r*, te se dodatno sva tri pokazivača pomjeraju za jedan čvor (linije 11-16). Na kraju se još pokazivač *kraj* ažurira tako da pokazuje na prethodno čelo liste (linija 17), dok se pokazivač *pocetak* postavlja na vrijednost tekućeg pokazivača *q* (linija 18), jer nakon izlaska iz **while** petlje, pokazivač *q* pokazuje na prethodno začelje liste.

Tabela 4.9

```
    OBRNI-POREDAK (L)
 1  p = L.pocetak
 2  q = NIL
 3  if (L.lduzina == 0) then
 4     L.tekuci = L.pocetak
 5  else if (L.dduzina == 0) then
 6     L.tekuci = NIL
 7  else
 8     L.tekuci = L.tekuci.sljedeci
 9  end_if
10  L.dduzina ↔ L.lduzina
11  while (p ≠ NIL) do
12     r = q
13     q = p
14     p = p.sljedeci
15     q.sljedeci = r
16  end_while
17  L.kraj = L.pocetak
18  L.pocetak = q
```

Obrtanje poretka čvorova u listi je ilustrirano na slici 4.13. Početno stanje povezane liste je prikazano na slici 4.13*a*, dok su koraci pri izvođenju operacije obrtanja prikazani na slikama 4.13*b*-*e*. Početna konfiguracija liste (5 | 12, 8, 9) se, nakon izvođenja operacije, pretvara u konfiguraciju (9, 8, 12 | 5). Prema tome, osim obrtanja poretka elemenata, također su međusobno razmijenjene i vrijednosti broja elemenata u lijevoj (*lduzina*) i desnoj (*dduzina*) particiji.



Slika 4.13. Obrtanje poretka elemenata u listi

4.2.7 Primjer implementacije jednostruko povezane liste u jeziku C++

U ovom primjeru je prikazana implementacija jednostruko povezane liste. Programski kod je organiziran u sljedećim datotekama:

- `lista.h`
- `jplista.h`
- `jplistatmp.h`
- `main.cpp`

Sadržaj datoteteke `lista.h` je već prethodno prikazan u primjeru 3.2.1, tako da je na ovom mjestu izostavljen. U datoteci `jplista.h` je definirana klasa `JPLista`, koja se derivira iz klase `Lista`. Sadržaj datoteke `jplista.h` je sljedeći:

jplista.h

```
// Definicija klase "JPLista"
#ifndef JPLISTA_H
#define JPLISTA_H

#include "lista.h"

template <typename InfoTip>
class JPLista : public Lista<InfoTip> {
private:
  struct Cvor {                          // Čvor
    InfoTip element;                     // Element liste
    Cvor *sljedeci;                      // Pokazivač na sljedbenika
    Cvor (const InfoTip &element, Cvor *sljedeci):
    element(element), sljedeci(sljedeci) {}
  };
  int lduzina,dduzina;              // Dužine lijeve i desne particije
  Cvor *pocetak;                    // Pokazivač na početak liste
  Cvor *kraj;                       // Pokazivač na kraj liste
  Cvor *tekuci;                     // Pokazivač na prethodnika tekućeg el.
  void Unisti();                    // Uništavanje liste
  void Iniciraj();                  // Inicijalizacija liste
public:
  JPLista() : pocetak(0),kraj(0),              // Konstruktor
  tekuci(0), lduzina(0),dduzina(0){}
  JPLista(const JPLista &jplista);             // Konstruktor kopije
  JPLista &operator =(const JPLista &jplista); // Preklopljeni op. dodjele
  ~JPLista();                                  // Destruktor
  void Brisi() {Unisti(); Iniciraj(); }  // Brisnaje liste
  bool Umetni(const InfoTip &element);   // Umetanje
  bool Dodaj(const InfoTip &element);    // Dodavanje
  InfoTip Izbaci();                      // Izbacivanje
  void IdiNaPocetak() {                  // Pozicioniranje na početak
    tekuci = 0;
    dduzina += lduzina;
    lduzina = 0;
  }
  void IdiNaKraj() {                     // Pozicioniranje na kraj
    tekuci = kraj;
    lduzina += dduzina;
    dduzina = 0;
  }
  void IdiNaPrethodni();                 // Pozicioniranje na prethodnika
  void IdiNaSljedeci();                  // Pozicioniranje na sljedbenika
  void IdiNaPoziciju(int i);             // Pozicioniranje na i-tu poz.
  int LDuzina() const { return lduzina; }      // Dužina lijeve particije
  int DDuzina() const { return dduzina; }      // Dužina desne particije
  bool Trazi(const InfoTip &x) const;          // Pretraživanje
  bool JeLiPrazna() const {               // Provjera je li lista prazna
    return lduzina + dduzina == 0;
  }
  int Duzina() const {                         // Vraća dužinu liste
    return lduzina + dduzina;
  }
  const InfoTip& DajTekuciElement() const;     // Vraća tekući el.
  InfoTip &operator [](int i);           // Preklopljeni op. indeksiranja
  InfoTip operator [] (int i) const;     // Preklopljeni op. indeksirabnja
  JPLista &operator +=(const JPLista &jplista);  // Preklopljeni operatori
  JPLista operator +(const JPLista &jplista) const;
  void Prikazi() const;                          // Prikaz sadržaja liste
```

```
 };

#endif
```

Klasa `JPLista` sadrži definiciju strukture `Cvor` koja modelira čvor liste. Osim toga, klasa `JPLista` sadrži pokazivače na početak liste (`pocetak`), kraj liste (`kraj`) i prethodnika tekućeg elementa (`tekuci`). Nadalje, klasa sadrži metode definirane apstraktnom klasom `Lista`, kao i nekoliko dodatnih metoda u privatnom i javnom dijelu. Privatne metode `Iniciraj` i `Unisti` služe za inicijalizaciju, odnosno uništavanje sadržaja liste. Ostale dodane metode su u javnom dijelu. Metoda `Trazi` služi za pretraživanje liste. Zatim, u listi se nalaze dvije varijante preklopljenog operatora `[]`, koje omogućuju označavanje pristupa elementima liste na sličan način kao što se označava pristup elementima niza. Naime, ove operatorske funkcije omogućuju da se elementu liste, koji ima indeks pozicije npr. 3, pristupi kao `lista[3]`, pri čemu prva varijanta omogućuje da se vraćeni element nađe s lijeve strane u izrazima dodjele, jer je vraćena referenca. Ako se operator poziva nad konstantnom listom, tada se poziva konstantna varijanta operatorske funkcije, koja vraća kopiju elementa kojem se pristupa, pa se ne može koristiti s lijeve strane u izrazima dodjele.

U datoteci `jplistatmp.h` su date definicije metoda klase `JPLista`. Sadržaj datoteke `jplistatmp.h` je sljedeći:

`jplistatmp.h`

```
// Definicija metoda klase "JPLista"
#ifndef JPLISTATMP_H
#define JPLISTATMP_H

#include "jplista.h"

template <typename InfoTip>            // Konstruktor kopije
JPLista<InfoTip>::JPLista(const JPLista &jplista) :
pocetak(0), kraj(0),dduzina(0),lduzina(0){
  Cvor *p(jplista.pocetak);
  while (p != 0) {
    Dodaj(p->element);
    p=p->sljedeci;
  }
}

template <typename InfoTip>            // Destruktor
JPLista<InfoTip>::~JPLista() {
  IdiNaPoziciju(0);
  while(!JeLiPrazna())
    Izbaci();
  pocetak = kraj = tekuci = 0;
}

template <typename InfoTip>            // Preklopljeni op. dodjele
JPLista<InfoTip> &JPLista<InfoTip>
::operator = (const JPLista &jplista) {
  if (&jplista==this)
    return *this;
  IdiNaPoziciju(0);
  while(!JeLiPrazna())
    Izbaci();
  Cvor *p(jplista.pocetak);
```

```
  while(p != 0) {
    Dodaj(p->element);
    p=p->sljedeci;
  }
  return *this;
}

template<typename InfoTip>              // Inicijalizacija liste
void JPLista<InfoTip>::Iniciraj() {
  tekuci = kraj = pocetak = 0;
  lduzina = dduzina = 0;
}

template<typename InfoTip>              // Uništavanje liste
void JPLista<InfoTip>::Unisti() {
  while(pocetak != 0) {
    tekuci = pocetak;
    pocetak = pocetak->sljedeci;
    delete tekuci;
  }
  pocetak = tekuci = kraj = 0;
}

template <typename InfoTip>             // Umetanje
bool JPLista<InfoTip>::Umetni(const InfoTip &element) {
  Cvor* p=new Cvor(element,0);
  if (lduzina == 0) {
    p->sljedeci = pocetak;
    pocetak = p;
    if (lduzina + dduzina == 0)
      kraj = p;
  }
  else {
    p->sljedeci = tekuci->sljedeci;
    tekuci->sljedeci = p;
    if (dduzina == 0)
      kraj = tekuci->sljedeci;
  }
  dduzina++;
  return true;
}

template <typename InfoTip>             // Dodavanje elementa
bool JPLista<InfoTip>::Dodaj(const InfoTip &element) {
  Cvor *p = new Cvor(element,0);
  if (lduzina+dduzina == 0)
    pocetak = kraj = p;
  else
    kraj = kraj->sljedeci = p;
  dduzina++;
  return true;
}

template <typename InfoTip>             // Izbacivanje
InfoTip JPLista<InfoTip>::Izbaci() {
  if (dduzina <= 0)
    throw "Nista za izbaciti!\n";
  InfoTip x;
  Cvor* priv;
  if (lduzina == 0) {
```

```
    x = pocetak->element;
    priv = pocetak;
    pocetak = priv->sljedeci;
  }
  else {
    priv = tekuci->sljedeci;
    x = tekuci->sljedeci->element;
    tekuci->sljedeci=priv->sljedeci;
  }
  if (dduzina == 1)
    kraj = tekuci;
  delete priv;
  dduzina--;
  return x;
}

template<typename InfoTip>              // Pozicioniranje na prethodnika
void JPLista<InfoTip>::IdiNaPrethodni() {
  if(lduzina != 0) {
    if (lduzina == 1)
      tekuci = 0;
    else {
      Cvor* p(pocetak);
      while (p->sljedeci != tekuci)
        p=p->sljedeci;
      tekuci = p;
    }
    lduzina--;
    dduzina++;
  }
}

template<typename InfoTip>              // Pozicioniranje na sljedbenika
void JPLista<InfoTip>::IdiNaSljedeci() {
  if (dduzina != 0) {
    if (lduzina == 0)
      tekuci = pocetak;
    else
      tekuci = tekuci->sljedeci;
    lduzina++;
    dduzina--;
  }
}

template <typename InfoTip>             // Pozicioniranje na i-tu poz.
void JPLista<InfoTip>::IdiNaPoziciju(int i) {
  if ((i<0) || (i>dduzina+lduzina))
    throw "Indeks izvan raspona!\n";
  int p(0);
  int ld(lduzina);
  dduzina = dduzina+lduzina-i;
  lduzina = i;
  if (i == 0) {
    tekuci = 0;
    return;
  }
  if (i >= ld) {
    if (ld == 0)
      tekuci = pocetak;
    else
```

```
      p = ld-1;
  }
  else
    tekuci = pocetak;
  for (int j = p; j <i-1; j++)
    tekuci = tekuci->sljedeci;
}

template <typename InfoTip>            // Pretraživanje
bool JPLista<InfoTip>::Trazi(const InfoTip &x) const {
  if (lduzina + dduzina == 0)
    throw "GRESKA:Lista prazna!\n";
  bool nadjen(false);
  Cvor *p(pocetak);
  while (p != 0 && p->element != x )
    p = p->sljedeci;
  if (p != 0)
    nadjen = true;
  return nadjen;
}

template <typename InfoTip>            // Vraća tekući element
const InfoTip& JPLista<InfoTip>::DajTekuciElement() const {
  if (dduzina == 0)
    throw "Nista za vratiti!\n";
  if (tekuci == 0)
    return pocetak->element;
  else
    return tekuci->sljedeci->element;
}

template <typename InfoTip>            // Operator pristupa
InfoTip &JPLista<InfoTip>::operator [](int i) {
  IdiNaPoziciju(i);
  if (i==0 || tekuci == 0)
    return pocetak->element;
  else
    return tekuci->sljedeci->element;
}

template <typename InfoTip>            // Konstantni operator pristupa
InfoTip JPLista<InfoTip>::operator [](int i) const {
  IdiNaPoziciju(i);
  if (i == 0 || tekuci == 0)
    return pocetak->element;
  else
    return tekuci->sljedeci->element;
}

template <typename InfoTip>            // Operator dodavanja
JPLista<InfoTip> &JPLista<InfoTip>::operator +=(const JPLista& jplista) {
  JPLista l(jplista);
  if (pocetak == 0)
    pocetak=l.pocetak;
  else
    kraj->sljedeci = l.pocetak;
  kraj = l.kraj;
  dduzina += l.dduzina;
  lduzina += l.lduzina;
  l.pocetak = l.kraj = l.tekuci=0;
```

```
  l.dduzina = l.lduzina = 0;
  return *this;
}

template <typename InfoTip>            // Operator spajanja
JPLista<InfoTip> JPLista<InfoTip>::operator +(const JPLista& jplista) const
{
  JPLista l(*this);
  l += jplista;
  return l;
}

template <typename InfoTip>            // Prikaz sadržaja liste
void JPLista<InfoTip>::Prikazi() const {
  Cvor *p(pocetak);
  int i(0);
  cout << "\nSadrzaj liste:\n";
  while (p != 0) {
    cout << i << ":" << p->element;
    if (p == pocetak)
      cout << "<- pocetak";
    if (p == tekuci)
      cout << "<- tekuci,lduzina:" << lduzina;
    if (p == kraj)
      cout << "<- kraj";
    cout << endl;
    i++;
    p=p->sljedeci;
  }
  cout << "lduzina:" << lduzina << endl;
  cout << "dduzina:" << dduzina << endl;
}

#endif
```

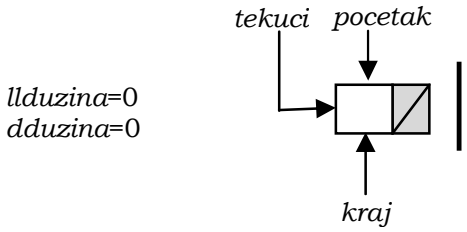
U datoteci `main.cpp` se nalazi programski kod testnog programa za jednostruko povezanu listu. Budući da je sadržaj ove datoteke skoro identičan sadržaju istoimene datoteke iz primjera 3.2.1, na ovom mjestu su navedene samo dvije linije koda, koje je potrebno izmjeniti:

	Linije koda koje treba izmjeniti		Novi sadržaj izmjenjenih linija
1	`#include "nizlistatmp.h"`	→	`#include "jplistatmp.h"`
2	`NizLista<int> lista;`	→	`JPLista<int> lista;`

4.3 Jednostruko povezane liste sa fiktivnim čvorom

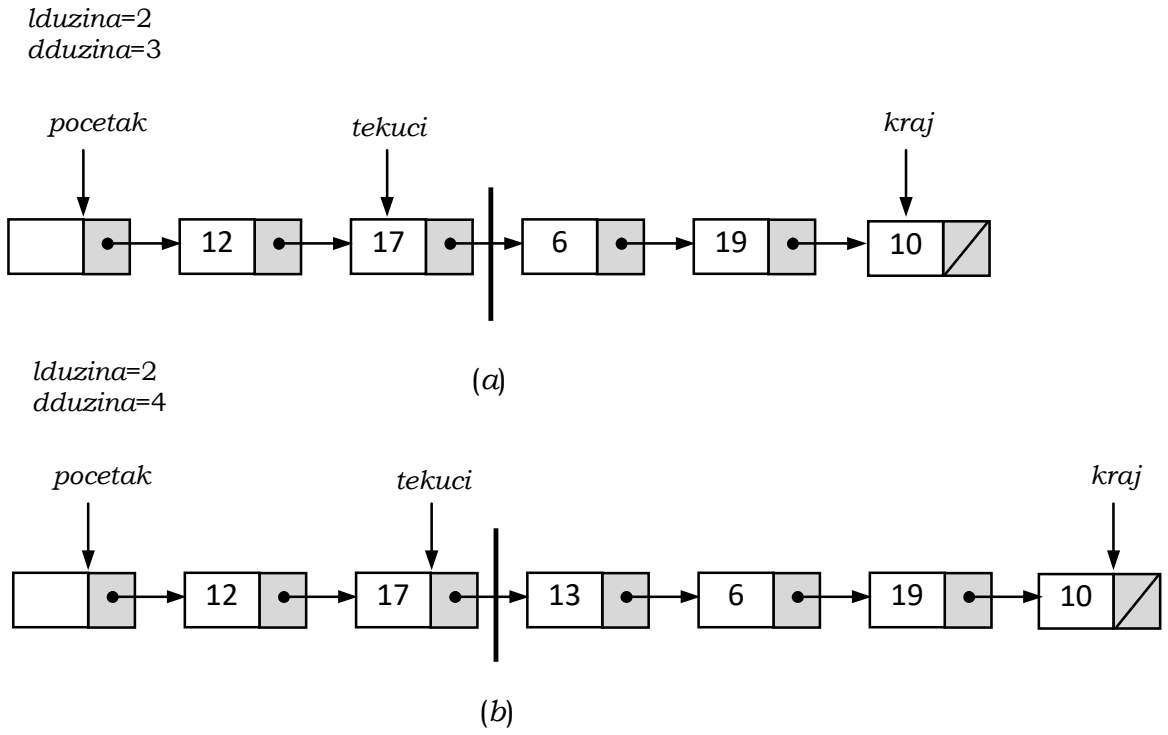
Iz prethodno opisanog pristupa implementaciji jednostruko povezanih listi, možemo vidjeti da su se pri implemenataciji pojedinih operacija morali uzimati u obzir specijalni slučajevi, koji se pojavljuju **kada je lista prazna**. To je za posljedicu imalo **povećanje kompleksnosti** tih operacija. Pri implementaciji povezane liste, specijalni slučajevi se mogu eliminirati uvođenjem tzv. **fiktivnog** čvora kao **čelnog čvora u listi**. Informacioni sadržaj fiktivnog čvora se zanemaruje i ne smatra se u logičkom smislu stvarnim

elementom liste. Dakle, uvođenjem fiktivnog čvora se **pojednostavljuju implementacije operacija**, jer neće biti potrebno uzimati u obzir specijalne slučajeve koji nastaju **kada je lista prazna, te kada je lijeva particija prazna**. Dakako, cijena uvođenja fiktivnog čvora je gubitak memorijskog prostora za jedan čvor. Međutim, bitno je istaknuti i to da se smanjenjem kompleksnosti implementiranih operacija, praktično najčešće uštedi više memorijskog prostora, nego što je dodatno potrebni prostor za jedan čvor. Moguća ušteda ili eventualni gubitak zapravo ovise i od broja listi koje se kreiraju. Na slici 4.14 je prikazano stanje inicijalizirane prazne liste, uz korištenje fiktivnog čvora.



Slika 4.14. Početno stanje prazne liste sa fiktivnim čvorom

Na slici 4.15 je ilustriran primjer umetanja novog čvora sa sadržajem x=13 u listu sa fiktivnim čvorom. Ovom operacijom se početna konfiguracija liste (12, 17 | 6, 19, 10) pretvara u konfiguraciju (12, 17 | 13, 6, 19, 10). Kao i kod liste bez fiktivnog čvora, umetanje se obavlja na početak desne particije, a pokazivač *tekuci* pokazuje na zadnji element lijeve particije.



Slika 4.15. Umetanje novog čvora u listu s fiktivnim čvorom

Budući da su operacije kod povezane liste s fiktivnim čvorom pojednostavljene, te da su analogne operacije kod listi bez fiktivnog čvora prethodno već detaljno opisane i ilustrirane, u nastavku je dat opis operacija u sažetijem obliku.

Operacija pretraživanja je identična kao i odgovarajuća operacija kod liste bez fiktivnog čvora, koja je prikazana u tabeli 4.8, s jedinom razlikom da se u liniji 4, umjesto naredbe *p = L.pocetak*, stavi naredba *p = L.pocetak.sljedeci*, da bi pretraživanje započelo od prvog stvarnog čvora, a ne od fiktivnog čvora.

Operacije pristupa sljedećem i prethodnom čvoru su opisane procedurama IDI-NA-SLJEDECI i IDI-NA-PRETHODNI u tabelama 4.10 i 4.11, respektivno. U odnosu na analogne procedure kod liste bez fiktivnog čvora, ovdje su procedure pojednostavljene. Naime, kod pristupa sljedećem čvoru, nije potrebno posebno obrađivati slučaj kada je lijeva particija prazna, dok kod pristupa prethodniku, nije potrebno posebno obrađivati slučaj kada lijeva particija ima samo jedan čvor.

Tabela 4.10

```
    IDI-NA-SLJEDECI (L)
1   if (L.dduzina ≠ 0]) then
2      L.tekuci = L.tekuci.sljedeci
3      L.lduzina = L.lduzina + 1
4      L.dduzina = L.dduzina - 1
5   end_if
```

Tabela 4.11

```
    IDI-NA-PRETHODNI (L)
1   if (L.lduzina ≠ 0) then
2      p = L.pocetak
3      while (p.sljedeci ≠ L.tekuci) do
4         p = p.sljedeci
5      end_while
6      L.tekuci = p
7      L.lduzina = L.lduzina -1
8      L.dduzina = L.dduzina + 1
9   end_if
```

Operacija, kojom se pristupa nekom elementu na poziciji s indeksom *i*, je opisana procedurom IDI-NA-POZICIJU u tabeli 4.12. Usporedbom sa procedurom prikazanom u tabeli 4.3, vidimo da nismo morali uzimati u obzir slučaj kada se pristupa poziciji *i*=0.

U tabeli 4.13 je opisana modificirana varijanta operacije pristupa, u obliku procedure IDI-NA-POZICIJU-M. Dakako, i u ovom slučaju možemo vidjeti da je procedura pojednostavljena, jer nismo trebali posebno tretirati situaciju kada se pristupa poziciji s indeksom *i*=0, kao i slučaj kada je lijeva particija prazna.

Tabela 4.12

```
    IDI-NA-POZICIJU (L, i)
1   if (i < 0 and i > L.lduzina + L.dduzina) then
2      ERROR („Indeks izvan dozvoljenog raspona“)
3   end_if
4   L.dduzina = L.dduzina + L.lduzina - i
5   L.lduzina = i
6   L.tekuci = L.pocetak
7   for j = 0 to i -1 do
8      L.tekuci = L.tekuci.sljedeci
9   end_for
```

Tabela 4.13

```
    IDI-NA-POZICIJU-M (L, i)
1   if (i < 0 and i > L.lduzina + L.dduzina) then
2      ERROR („Indeks izvan dozvoljenog raspona“)
3   end_if
4   L.dduzina = L.dduzina + L.lduzina - i
5   L.lduzina = i
6   p = 0
7   ld = L.lduzina
8   if (i ≥ ld) then
9      p = ld
10  else
11     L.tekuci = L.pocetak
12  end_if
13  for j = p to i -1 do
14     L.tekuci = L.tekuci.sljedeci
15  end_for
```

Operacija umetanja je opisana u tabeli 4.14 procedurom UMETNI. Zbog toga što nije potrebno posebno uzimati u obzir slučaj kada je lijeva particija liste prazna, procedura je pojednostavljena u odnosu na odgovarajuću proceduru kod implementacije liste bez fiktivnog čvora.

Tabela 4.14

```
    UMETNI (L, x)
1   p = GETNODE ()
2   p.info = x
3   p.sljedeci = L.tekuci.sljedeci
4   L.tekuci.sljedeci = p
5   if ( L.kraj == L.tekuci) then
6      L.kraj = p
7   end_if
8   L.dduzina = L.dduzina + 1
```

Operacija dodavanja novog čvora na kraj liste je opisana procedurom DODAJ u tabeli 4.15. Naravno, i u ovom slučaju je operacija pojednostavljena, jer se nije trebao posebno tretirati slučaj kada je lista prazna.

Tabela 4.15

```
    DODAJ (L, x)
1   p = GETNODE ()
2   p.info = x
3   p.sljedeci = NIL
4   L.kraj = L.kraj.sljedeci = p
5   L.dduzina = L.dduzina + 1
```

U tabeli 4.16 je opisana operacija izbacivanja iz liste u obliku funkcije IZBACI, koja kao i funkcija IZBACI u tabeli 4.7 vraća informacioni sadržaj čvora koji se izbacuje. Opet je implementacija funkcije pojednostavljena, jer se nije trebao posebno tretirati slučaj kada je lijeva particija liste prazna.

Tabela 4.16

```
    IZBACI (L)
1   if ( L.dduzina == 0 ) then
2      ERROR („Nista za izbaciti!“)
3   end_if
4   x = L.tekuci.sljedeci.info
5   p = L.tekuci.sljedeci
6   L.tekuci.sljedeci = p.sljedeci
7   if (p == L.kraj) then
8      L.kraj = L.tekuci
9   end_if
10  FREENODE (p)
11  L.dduzina = L.dduzina + 1
12  return x
```

4.3.1 Primjer implementacije jednostruko povezane liste sa fiktivnim čvorom u jeziku C++

U ovom primjeru je prikazana implementacija jednostruko povezane liste s fiktivnim čvorom. Programski kod je organiziran u sljedeće datoteke:

- `lista.h`
- `jplistafc.h`
- `jplistafctmp.h`
- `main.cpp`

Sadržaj datoteteke `lista.h` je već prethodno prikazan (primjer 3.2.1), tako da je na ovom mjestu izostavljen. U datoteci `jplistafc.h` je definirana klasa `JPListaFC` koja se derivira iz klase `Lista`. Klasa `JPListaFC` sadrži definiciju strukture `Cvor` koja modelira čvor liste, kao i pokazivače na početak liste (`pocetak`), kraj liste (`kraj`) i prethodnika tekućeg elementa (`tekuci`). Međutim, za razliku od primjera 4.2.7, koji je

ilustrirao implementaciju liste bez fiktivnog čvora, na ovom mjestu su iz praktičnih razloga izostavljeni konstruktor kopije i preklopljeni operator dodjele. Osim toga, od dodatnih metoda, u odnosu na apstraktnu klasu `Lista`, ovdje je jedino prikazana metoda za prikaz sadržaja liste. Sadržaj datoteke `jplistafc.h` je sljedeći:

jplistafc.h

```
// Definicija klase "JPListaFC"
#ifndef JPLISTAFC_H
#define JPLISTAFC_H

#include "lista.h"

template <typename InfoTip>
class JPListaFC : public Lista<InfoTip> {
private:
  struct Cvor {              // Čvor u listi
    InfoTip element;         // Element liste
    Cvor *sljedeci;          // Pokazivač na sljedeći element liste
    Cvor (const InfoTip &element, Cvor *sljedeci=0):
    element(element), sljedeci(sljedeci) {}
    Cvor(Cvor* sljedeci=0):sljedeci(sljedeci) {}
  };
  Cvor* pocetak;             // Pokazivač na početak liste
  Cvor* kraj;                // Pokazivač na kraj liste
  Cvor* tekuci;              // Pokazivač na prethodnika tekućeg elementa
  int lduzina;               // Dužina lijeve particije
  int dduzina;               // Dužina desne particije
  void Iniciraj();           // Inicijalizacija liste
  void Unisti();             // Uništavanje liste
public:
  JPListaFC (int velicina = 10) { Iniciraj(); }     // Konstruktor
  ~JPListaFC() { Unisti(); }                        // Destruktor
  void Brisi() { Unisti(); Iniciraj(); }            // Brisanje liste
  bool Umetni(const InfoTip& x);                   // Umetanje
  bool Dodaj(const InfoTip& x);                    // Dodavanje
  InfoTip Izbaci();                                 // Izbacivanje
  void IdiNaPocetak() {                             // Pozicioniranje na početak
    tekuci = pocetak;
    dduzina += lduzina;
    lduzina = 0;
  }
  void IdiNaKraj() {                                // Pozicioniranje na kraj
    tekuci = kraj;
    lduzina += dduzina;
    dduzina = 0;
  }
  void IdiNaPrethodni();                 // Pozicioniranje na prethodni el.
  void IdiNaSljedeci();                  // Pozicioniranje na sljedeći el.
  void IdiNaPoziciju(int i);                 // Pozicioniranje na i-tu poz.
  int LDuzina() const { return lduzina; }  // Vraća dužinu lijeve part.
  int DDuzina() const { return dduzina; }  // Vraća dužinu desne part.
  const InfoTip& DajTekuciElement() const; // Vraća tekući element
  void Prikazi() const;                    // Prikazuje sadržaj liste
};

#endif
```

U datoteci jplistafctmp.h su date implementacije metoda klase JPLista. Možemo primijetiti da je implementacija tih metoda prilično jednostavnija u odnosu na implementaciju analognih metoda kod liste bez fiktivnog čvora. Sadržaj datoteke jplistafctmp.h je sljedeći:

jplistafctmp.h

```
// Definicija metoda klase JPListaFC
#ifndef JPLISTAFCTMP_H
#define JPLISTAFCTMP_H

#include "jplistafc.h"

template<typename InfoTip>                        // Inicijalizacija liste
void JPListaFC<InfoTip>::Iniciraj() {
  tekuci= kraj = pocetak = new Cvor;
  lduzina = dduzina = 0;
}

template<typename InfoTip>                        // Uništavanje liste
void JPListaFC<InfoTip>::Unisti() {
  while(pocetak != 0) {
    tekuci = pocetak;
    pocetak = pocetak->sljedeci;
    delete tekuci;
  }
  pocetak = 0; kraj = 0; tekuci = 0;
}

template<typename InfoTip>                        // Umetanje
bool JPListaFC<InfoTip>::Umetni(const InfoTip& x) {
  tekuci->sljedeci = new Cvor(x, tekuci->sljedeci);
  if (kraj == tekuci)
    kraj = tekuci->sljedeci;
  dduzina++;
  return true;
}

template<typename InfoTip>                        // Dodavanje
bool JPListaFC<InfoTip>::Dodaj(const InfoTip& x) {
  kraj=kraj->sljedeci = new Cvor(x, 0);
  dduzina++;
  return true;
}

template<typename InfoTip>                        // Izbacivanje
InfoTip JPListaFC<InfoTip>::Izbaci() {
  if(dduzina == 0)
    throw "Nista za obrisati!\n";
  InfoTip x = tekuci->sljedeci->element;
  Cvor* lpriv = tekuci->sljedeci;
  tekuci->sljedeci = lpriv->sljedeci;
  if (kraj == lpriv) kraj = tekuci;
  delete lpriv;
  dduzina--;
  return x;
}
```

```
template<typename InfoTip>              // Pozicioniranje na prethodnika
void JPListaFC<InfoTip>::IdiNaPrethodni() {
  if (lduzina!=0) {
    Cvor* priv = pocetak;
    while (priv->sljedeci != tekuci)
      priv = priv->sljedeci;
    tekuci = priv;
    lduzina--;
    dduzina++;
  }
}

template<typename InfoTip>              // Pozicioniranje na sljedbenika
void JPListaFC<InfoTip>::IdiNaSljedeci() {
  if (dduzina != 0) {
    tekuci = tekuci->sljedeci;
    dduzina--;
    lduzina++;
  }
}

template<typename InfoTip>              // Pozicioniranje na i-tu poz.
void JPListaFC<InfoTip>::IdiNaPoziciju(int i) {
  if ((i < 0) || (i > dduzina + lduzina))
    throw "Indeks izvan dozvoljenog raspona!\n";
  int p(0),ld(lduzina);
  dduzina = dduzina + lduzina - i;
  lduzina = i;
  if (i >= ld)
    p = ld;
  else
    tekuci = pocetak;
  for (int j = p; j < i; j++)
    tekuci = tekuci->sljedeci;
}

template<typename InfoTip>              // Vraća tekući element
const InfoTip& JPListaFC<InfoTip>::DajTekuciElement() const {
  if (dduzina == 0)
    throw "Nista za vratiti!\n";
  return tekuci->sljedeci->element;
}

template<typename InfoTip>              // Prikazuje sadržaj liste
void JPListaFC<InfoTip>::Prikazi() const {
  if (dduzina == 0)
    throw "Nista za prikazati!\n";
  Cvor* priv = pocetak->sljedeci;
  while ( priv!= 0 ) {
    cout << priv->element << endl;
    priv = priv->sljedeci;
  }
}

#endif
```

U datoteci `main.cpp` se nalazi programski kod testnog programa za jednostruko povezanu listu sa fiktivnim čvorom. Dakako, sadržaj i ove datoteke (slično kao i u primjeru

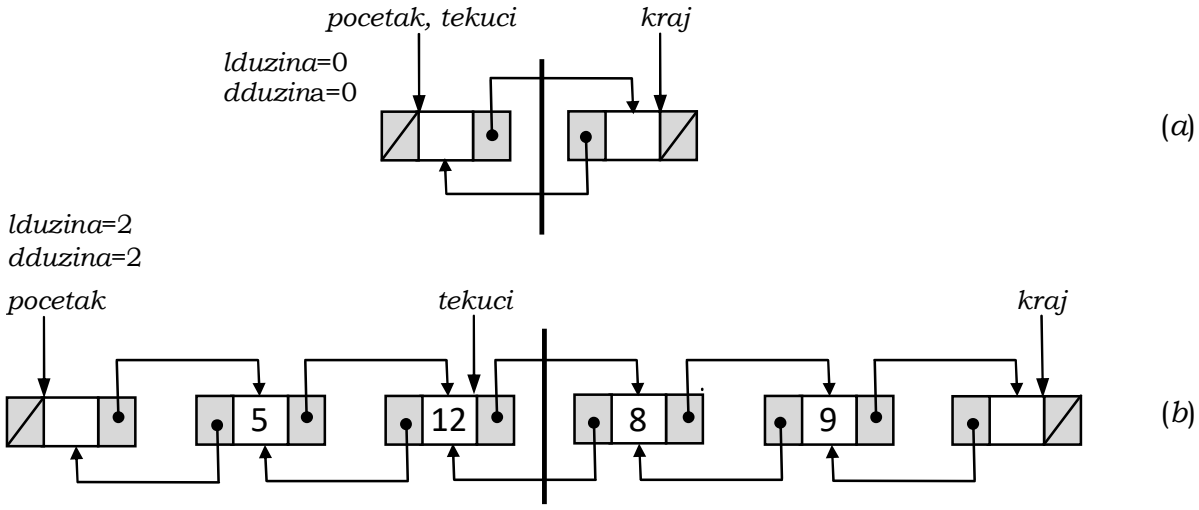
4.2.7) je skoro identičan sadržaju istoimene datoteke iz primjera 3.2.1, pa su na ovom mjestu navedene samo dvije linije koda koje je potrebno izmjeniti:

	Linije koda koje treba izmjeniti		**Novi sadržaj izmjenjenih linija**
1	`#include "nizlistatmp.h"`	→	`#include "jplistafctmp.h"`
2	`NizLista<int> lista;`	→	`JPListaFC<int> lista;`

4.4 Dvostruko povezane liste

Prethodno smo vidjeli da se kod jednostruko povezanih listi, direktni pristup nekom čvoru može ostvariti samo preko njegovog prethodnika. Isto tako, direktni pristup se može ostvariti prvom čvoru u listi, zadnjem čvoru lijeve particije i zadnjem čvoru liste, zahvaljući pokazivačima *pocetak*, *tekuci* i *kraj*, respektivno. Međutim, takav pristup elementima, kao i kretanje kroz listu je neefikasno. Na primjer, ako je potrebno preći od nekog čvora do njegovog prethodnika, mora se preći cijeli dio liste počevši od čela do prethodnika. Jedan od načina kako možemo povećati efikasnost kretanja kroz povezanu listu je da svaki čvor pored pokazivača na sljedeći čvor (*sljedeci*), sadrži i pokazivač na prethodni čvor (*prethodni*). Na taj način ćemo dobiti **dvostruko povezanu listu**. Slično kao i kod jednostruko povezanih listi, da bismo pojednostavili implementacije operacija, mogu se koristiti fiktivni čvorovi, pri čemu se osim **čelnog fiktivnog čvora** može uvesti i **fiktivni čvor na začelju liste**. Na ovom mjestu ćemo se odlučiti upravo za varijantu sa fiktivnim čvorovima, dok se čitaocu preporučuje da svakako implementira i varijantu bez fiktivnih čvorova.

Na slici 4.16*a* je ilustrirana konfiguracija prazne dvostruko povezane liste, dok je na slici 4.16*b* prikazan primjer liste sa sljedećom konfiguracijom (5, 12 | 8, 9).



Slika 4.16. Dvostruko povezana lista

Sa dvostruko povezanim listama se mogu izvoditi sve prethodno opisane operacije, kao i sa jednostruko povezanim listama. Dakako, treba istaknuti da je pri obavljanju tipičnih operacija potrebno ažurirati oba pokazivača (*sljedeci* i *prethodni*).

Operacija pristupa sljedećem čvoru je identična kao i analogna operacija kod jednostruko povezanih listi. Međutim, za razliku od jednostruko povezane liste, operacija pristupa prethodniku se može ostvariti direktno, što je opisano procedurom IDI-NA-PRETHODNI u tabeli 4.17. Operacija ne proizvodi nikakav učinak, ako je lijeva particija prazna.

Tabela 4.17

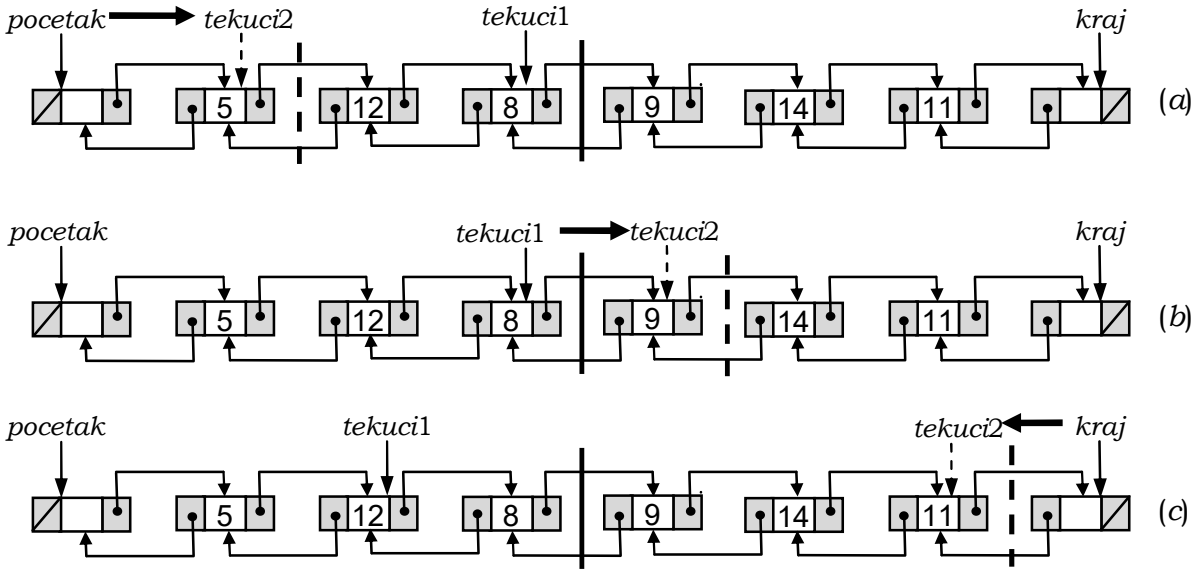
```
    IDI-NA-PRETHODNI (L)
1   if ( L.lduzina ≠ 0) then
2      L.tekuci = L.tekuci.prethodni
3      L.lduzina = L.lduzina - 1
4      L.dduzina = L.dduzina - 1
5   end_if
```

Tabela 4.18

```
    IDI-NA-POZICIJU-M (L, i)
1   if (i < 0 or i ≥ L.lduzina + L.dduzina) then
2      ERROR („Indeks izvan dozvoljenog raspona“)
3   end_if
4   ld = L.lduzina
5   dd = L.dduzina
6   L.dduzina = L.dduzina + L.lduzina - i
7   L.lduzina = i
8   if (i < |i-ld| and (i < (ld+dd)/2) then
9      k = i
10     L.tekuci = L.pocetak
11  else if (|i-ld| ≤ |ld+dd-i|) then
12     k = i-ld
13  else
14     k = i-ld-dd-1
15     L.tekuci = L.kraj
16  end_if
17  if (k < 0) then
18     for i = 0 to |k| do
19        p = L.tekuci.prethodni
20     end_for
21  else
22     for i = 0 to |k| do
23        p = L.tekuci.sljedeci
24     end_for
25  end_if
```

Osnovna varijanta operacije pristupa elementu na poziciji s indeksom *i*, je također ista kao i kod jednostruko povezane liste. Međutim, modificirana varijanta se dodatno može poboljšati zahvaljujući mogućnosti kretanja kroz listu u oba smjera. Naime, kod dvostruko povezane liste, pristup nekom elementu je najbolje ostvariti tako da pristup počne od početka, kraja ili tekuće pozicije, ovisno o tome koji čvor je najbliži poziciji kojoj se pristupa. Na slici 4.17 su ilustrirane tri moguće situacije za početnu konfiguraciju liste (5, 12, 8 | 9, 14, 11). Ako se, na primjer, pristupa poziciji i=1, tada se pristup najefikasnije ostvaruje u jednom koraku počevši od početnog čvora (slika 4.17*a*). S druge strane, ako se pristupa poziciji i=4, tada se pristup u jednom koraku ostvaruje iz tekućeg čvora (slika 4.17*b*). I konačno, ako se pristupa poziciji i=6, pristup samo u jednom koraku se ostvaruje iz krajnjeg čvora, koristeći vezu na prethodnika (slika 4.17*c*).

Modificirana varijanta pristupa, koja uzima u obzir prethodno opisanu mogućnost poboljšanja, je opisana procedurom IDI-NA-POZICIJU-M u tabeli 4.18. Ispitivanjem odgovarajućih uvjeta se utvrđuje iz koje pozicije se u najmanjem broju koraka k može ostvariti pristup (linije 8-16). Nakon toga se korištenjem veza na prethodnike (linije 18-20) ili na sljedbenike ostvaruje pristup u k koraka (linije 22-24).



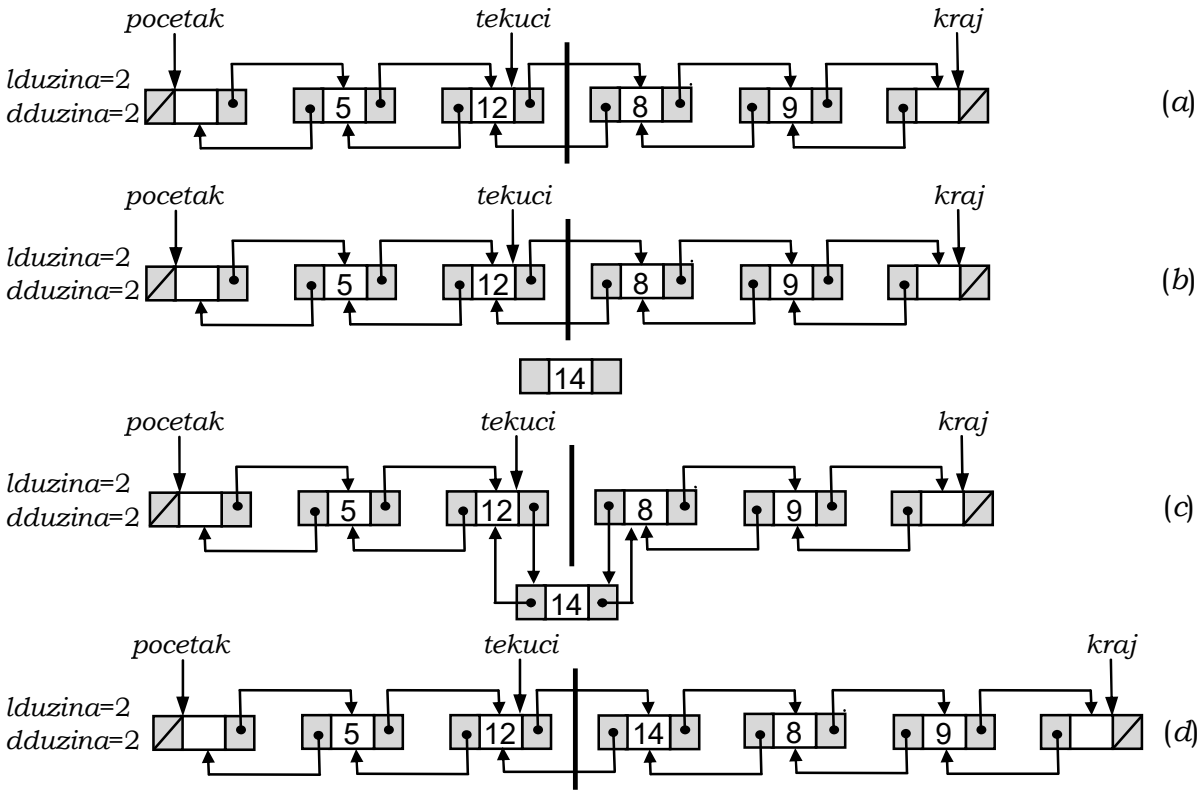
Slika 4.17. Pristup čvorovima dvostruko povezane liste

Operacija umetanja je opisana procedurom UMETNI u tabeli 4.19. Nakon kreiranja novog čvora i upisivanja informacionog sadržaja (linije 1-2), lista se prevezuje tako da novi čvor postaje sljedbenik tekućeg čvora (linije 3 i 5), te prethodnik bivšeg sljedbenika tekućeg čvora (linije 4 i 6). Na kraju se ažurira dužina desne particije (linija 7).

Tabela 4.19

```
    UMETNI (L, x)
1   p = GETNODE ()
2   p.info = x
3   p.prethodni = L.tekuci
4   p.sljedeci = L.tekuci.sljedeci
5   L.tekuci.sljedeci = p
6   p.sljedeci.prethodni = p
7   L.dduzina = L.dduzina + 1
```

Operacija kojom se početna konfiguracija liste (5, 12 | 8, 9), umetanjem elementa 14, pretvara u konfiguraciju (5, 12 | 14, 8, 9), je ilustrirana na slici 4.18. Slika 4.18*a* prikazuje početno stanje, dok je na slici 4.18*b* prikazano stanje nakon kreiranja novog čvora i upisivanja sadržaja 14. Stanje nakon prevezivanja liste je prikazano na slici 4.18*c*, dok je na slici 4.18*d*, samo grafički drugačije prikazano stanje liste, uz ažuriranu dužinu desne particije *dduzina*=3.



Slika 4.18. Umetanje novog čvora u dvostruko povezanu listu

Operacija dodavanja novog čvora je opisana procedurom DODAJ u tabeli 4.20. Budući da je ova operacija vrlo slična prethodno opisanoj operaciji umetanja, pri čemu

ulogu tekućeg čvora preuzima prethodnik fiktivnog krajnjeg čvora, a ulogu sljedbenika tekućeg čvora preuzima sam krajnji fiktivni čvor.

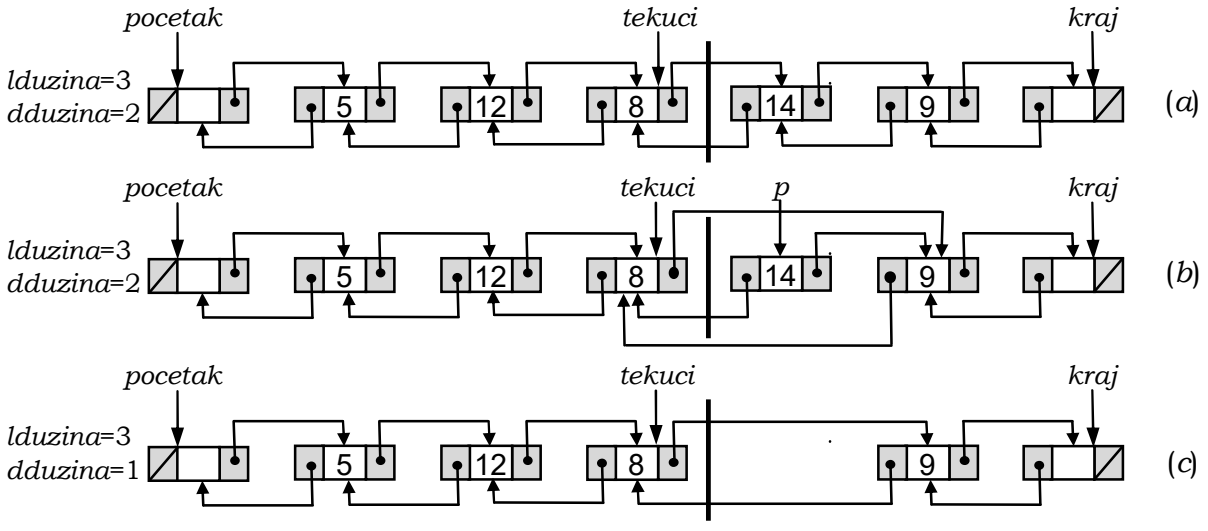
Tabela 4.20

```
    DODAJ (L, x)
1   p = GETNODE ()
2   p.info = x
3   p.prethodni = L.kraj.prethodni
4   p.sljedeci = L.kraj
5   L.kraj.prethodni.sljedeci = p
6   L.kraj.prethodni = p
7   L.dduzina = L.dduzina + 1
```

Operacija izbacivanja je opisana procedurom IZBACI u tabeli 4.21. U privremenom pokazivaču *p* se čuva adresa čvora koji se izbacuje, a zatim se lista prevezuje tako da tekući čvor postaje prethodnik sljedbenika čvora koji se izbacuje (linije 6 i 7). Na kraju se oslobađa memorijski prostor, koji je dotični čvor zauzimao, ažurira se dužina desne particije, tako da se smanjuje za 1, te se vraća vrijednost *x*, koja je prethodno bila smještena u čvoru koji se izbacuje (linija 10).

Tabela 4.21

```
    IZBACI (L)
1   if (L.dduzina == 0) then
2      ERROR („Nista za izbaciti“)
3   end_if
4   x = L.tekuci.sljedeci.info
5   p = L.tekuci.sljedeci
6   p.sljedeci.prethodni = L.tekuci
7   L.tekuci.sljedeci = p.sljedeci
8   FREENODE (p)
9   L.dduzina = L.dduzina - 1
10  return x
```



Slika 4.19. Izbacivanje čvora iz dvostruko povezane liste

Operacija izbacivanja je ilustrirana na slici 4.19, izbacivanjem čvora 14, pri čemu se početna konfiguracija (5, 12, 8 | 14, 9) pretvara u konfiguraciju (5, 12, 8 | 9). Početno stanje je prikazano na slici 4.19*a*. Na slici 4.19*b* je prikazano stanje nakon prevezivanja liste, dok je na slici 4.19*c* prikazano stanje nakon oslobađanja prostora koji je čvor zauzimao i ažuriranja dužine desne particije na *dduzina*=1.

4.4.1 Primjer implementacije dvostruko povezane liste u jeziku C++

U ovom primjeru je prikazana implementacija dvostruko povezane liste s fiktivnim čvorovima. Programski kod je organiziran u sljedećim datotekama:

- `lista.h`
- `dplistafc.h`
- `dplistafctmp.h`
- `main.cpp`

Sadržaj datoteteke `lista.h` je već prethodno prikazan (primjer 3.2.1), tako da je na ovom mjestu izostavljen. U datoteci `dplistafc.h` je definirana klasa `DPListaFC`, koja se kao i klasa `JPLista` također derivira iz klase `Lista`. Sadržaj datoteke `dplistafc.h` je sljedeći:

dplistafc.h

```
// Definicija klase "DPListaFC"
#ifndef DPLISTAFC_H
#define DPLISTAFC_H

#include "lista.h"

template <typename InfoTip>
class DPListaFC : public Lista<InfoTip> {
private:
  struct Cvor {                                  // Definicija čvora
    InfoTip element;                             // Element liste
    Cvor *sljedeci;                              // Pokazivač na sljedbenika
    Cvor *prethodni;                             // Pokazivač na prethodnika
    Cvor (const InfoTip &element, Cvor* prevp, Cvor *nextp=0):
    element(element), prethodni(prevp),sljedeci(nextp) {
      if ( prethodni != 0) prethodni->sljedeci=this;
      if (sljedeci != 0) sljedeci->prethodni=this;
    }
    Cvor(Cvor* prevp=0, Cvor* nextp=0):prethodni(prevp),sljedeci(nextp) {
      if (prethodni != 0 ) prethodni->sljedeci = this;
      if (sljedeci != 0) sljedeci->prethodni = this;
    }
  };
  Cvor* pocetak;                  // Pokazivač na fiktivni početni čvor
  Cvor* kraj;                     // Pokazivač na fiktivni krajnji čvor
  Cvor* tekuci;                   // Pokazivač na prethodnika tekućeg
  int lduzina;                    // Dužina lijeve particije
  int dduzina;                    // Dužina desne particije
  void Iniciraj();                // Inicijalizacija liste
  void Unisti();                  // Uništavanje liste
public:
  DPListaFC (int velicina=10) { Iniciraj(); }    // Konstruktor
```

```
  ~DPListaFC() { Unisti(); }                  // Destruktor
  void Brisi() { Unisti(); Iniciraj(); }      // Brisanje liste
  bool Umetni(const InfoTip& x);              // Umetanje
  bool Dodaj(const InfoTip& x);               // Dodavanje
  InfoTip Izbaci();                           // Izbacivanje
  void IdiNaPocetak() {                       // Pomjeranje na početak
    tekuci = pocetak;
    dduzina+=lduzina;
    lduzina = 0;
  }
  void IdiNaKraj() {                          // Pomjeranje na kraj
    tekuci = kraj;
    lduzina += dduzina;
    dduzina = 0;
  }
  void IdiNaPrethodni();                      // Pomjeranje na prethodnika
  void IdiNaSljedeci();                       // Pomjeranje na sljedbenika
  void IdiNaPoziciju(int i);                  // Pomjeranje na i-tu poz.
  int LDuzina() const { return lduzina; }     // Vraća dužinu lijeve p.
  int DDuzina() const { return dduzina; }     // Vraća dužinu desne p.
  const InfoTip& DajTekuciElement() const;    // Vraća tekući el.
  void Prikazi() const;                       // Prikazuje sadržaj liste
};

#endif
```

Klasa `DPListaFC` u privatnom dijelu sadrži definiciju strukture `Cvor`, koja modelira čvor dvostruko povezane liste. Čvor pored atributa `element` za smještanje elementa liste, sadrži pokazivače na sljedbenika (`sljedeci`) i prethodnika (`prethodni`), što omogućuje kretanje kroz listu u oba smjera. Nadalje, u privatnom dijelu klase su i pokazivači na početak liste (`pocetak`), kraj liste (`kraj`), kao i prethodnika tekućeg elementa (`tekuci`). U javnom dijelu klase se nalaze metode navedene u apstraktnoj klasi `Lista`, uz dodatnu metodu `Prikaz`, koja služi za prikaz sadržaja liste. U ovom primjeru su također izostavljeni konstruktori kopije, te preklopljeni operator dodjele.

U datoteci `dplistafctmp.h` su date implementacije metoda klase `DPListaFC`. Sadržaj datoteke `dplistafctmp.h` je sljedeći:

`dplistafctmp.h`

```
// Definicija metoda klase DPListaFC
#ifndef DPLISTAFCTMP_H
#define DPLISTAFCTMP_H

#include "dplistafc.h"

template<typename InfoTip>                    // Inicijalizacija
void DPListaFC<InfoTip>::Iniciraj() {
  tekuci=pocetak = new Cvor(0,0);
  pocetak->sljedeci = kraj = new Cvor (pocetak, 0);
  lduzina = dduzina = 0;
}
```

```
template<typename InfoTip>                    // Uništavanje liste
void DPListaFC<InfoTip>::Unisti() {
  while(pocetak!=0) {
    tekuci = pocetak;
    pocetak = pocetak->sljedeci;
    delete tekuci;
  }
  kraj = 0; tekuci = 0;
}

template<typename InfoTip>                    // Umetanje
bool DPListaFC<InfoTip>::Umetni(const InfoTip& x) {
  new Cvor(x, tekuci, tekuci->sljedeci);
  dduzina++;
  return true;
}

template<typename InfoTip>                    // Dodavanje
bool DPListaFC<InfoTip>::Dodaj(const InfoTip& x) {
  new Cvor(x, kraj->prethodni, kraj);
  dduzina++;
  return true;
}

template<typename InfoTip>                    // Izbacivanje
InfoTip DPListaFC<InfoTip>::Izbaci() {
  if(dduzina == 0)
    throw "Nista za obrisati!\n";
  InfoTip x = tekuci->sljedeci->element;
  Cvor* lpriv = tekuci->sljedeci;
  lpriv->sljedeci->prethodni = tekuci;
  tekuci->sljedeci = lpriv->sljedeci;
  delete lpriv;
  dduzina--;
  return x;
}

template<typename InfoTip>                    // Pomjeranje na prethodnika
void DPListaFC<InfoTip>::IdiNaPrethodni() {
  if (tekuci != pocetak) {
    tekuci = tekuci->prethodni;
    lduzina--;
    dduzina++;
  }
}

template<typename InfoTip>                    // Pomjeranje na sljedbenika
void DPListaFC<InfoTip>::IdiNaSljedeci() {
  if (tekuci != kraj) {
    tekuci = tekuci->sljedeci;
    dduzina--;
    lduzina++;
  }
}
```

```
template<typename InfoTip>                    // Pomjeranje na i-tu poz.
void DPListaFC<InfoTip>::IdiNaPoziciju(int i) {
  if ((i < 0) || (i > dduzina + lduzina))
    throw "Indeks izvan dozvoljenog raspona!\n";
  int p(0),ld(lduzina), dd (dduzina);
  dduzina = dduzina + lduzina - i;
  lduzina = i; int k(0);
  if (i < abs(i-ld) && i < (ld+dd)/2) {
    k=i;
    tekuci = pocetak;
  }
  else if (abs(i-ld) < ld+dd-i)
    k=i-ld;
  else {
    k = i-ld-dd-1;
    tekuci = kraj;
  }
  if (k<0)
    for (int i = 0; i < abs(k); i++)
      tekuci = tekuci->prethodni;
  else
    for (int i=0; i < k; i++)
      tekuci = tekuci->sljedeci;
 }


template<typename InfoTip>                    // Vraća tekući element
const InfoTip& DPListaFC<InfoTip>::DajTekuciElement() const {
  if (dduzina == 0)
    throw "Nista za vratiti!\n";
  return tekuci->sljedeci->element;
}

template<typename InfoTip>                    // Prikazuje sadržaj liste
void DPListaFC<InfoTip>::Prikazi() const {
  if (dduzina + lduzina == 0)
    throw "Nista za prikazati!\n";
  Cvor* p = pocetak->sljedeci;
  while ( p!= kraj ) {
    cout << p->element << endl;
    p = p->sljedeci;
  }
}

#endif
```

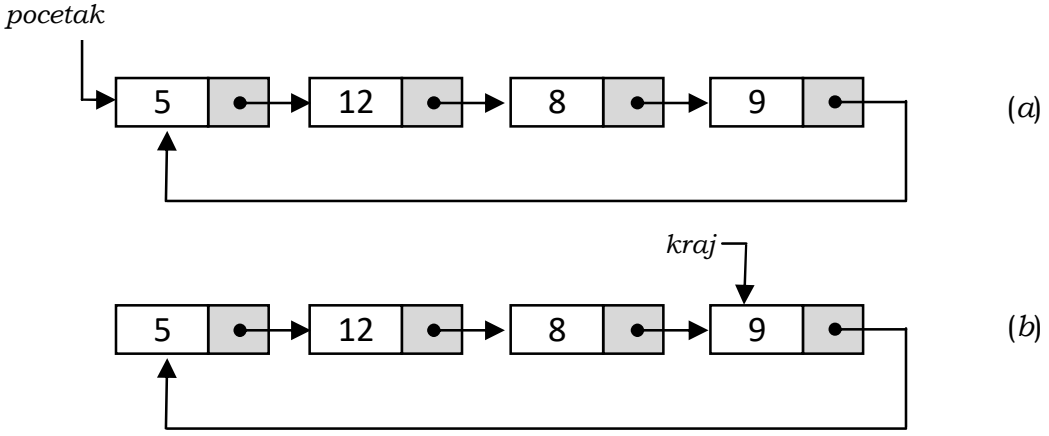
Možemo primijetiti da je implementacija metoda klase `DPListaFC` prilično jednostavnija u odnosu na implementaciju analognih metoda kod jednostruko povezane liste bez fiktivnog čvora, bez obzira što je, općenito govoreći, implementacija dvostruko povezane liste složenija, jer se moraju ažurirati pokazivači i na prethodnika i na sljedbenika.

U datoteci `main.cpp` se nalazi programski kod testnog programa za dvostruko povezanu listu sa fiktivnim čvorom. Sadržaj ove datoteke (slično kao i u primjerima 4.2.7 i 4.3.1) je skoro identičan sadržaju istoimene datoteke iz primjera 3.2.1, pa su na ovom mjestu navedene samo dvije linije koda koje je potrebno izmjeniti:

	Linije koda koje treba izmjeniti		Novi sadržaj izmjenjenih linija
1	`#include "nizlistatmp.h"`	→	`#include "dplistafctmp.h"`
2	`NizLista<int> lista;`	→	`DPListaFC<int> lista;`

4.5 Cirkularne liste

Jedan od nedostataka jednostruko povezane liste je ograničenje kretanja kroz listu isključivo u jednom smjeru, od čela prema začelju. Jedan od načina na koji se može eliminirati ovaj nedostatak je upotreba **cirkularne liste**. Cirkularnu listu možemo dobiti tako da u polje pokazivača *sljedeci* zadnjeg čvora u listi, umjesto vrijednosti NIL, upišemo pokazivač na prvi čvor u listi (slika 4.20*a*). Na taj način je omogućeno da se, iz bilo kojeg početnog čvora, može doći do bilo kojeg drugog čvora, neovisno od položaja početnog čvora.

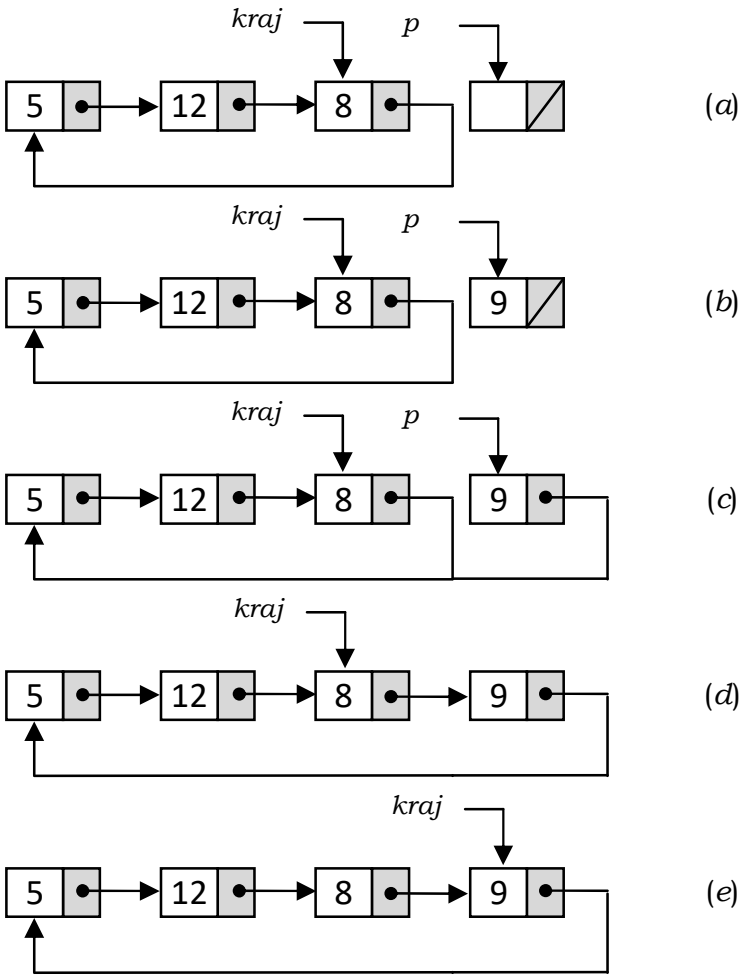


Slika 4.20. Cirkularna jednostruko povezana lista: *a*) vanjski pokazivač pokazuje na prvi čvor, *b*) vanjski pokazivač pokazuje na zadnji čvor

Osim toga, važno je primijetiti da su termini „*prvi čvor*“ i „*zadnji čvor*“ problematični u slučaju cirkularne povezane liste. Na slici 4.20*a* je prikazan primjer cirkularne liste kod koje se čvorovima pristupa preko vanjskog pokazivača *pocetak*, koji pokazuje na prvi čvor u listi, dok je na slici 4.20*b* ilustrirana cirkularna lista kod koje se čvorovima liste pristupa preko vanjskog pokazivača *kraj*, koji pokazuje na zadnji čvor u listi. Nadalje, analizom operacija umetanja novog čvora na početak i na kraj liste, te analizom operacija brisanja čvora sa početka i sa kraja liste, možemo doći do zaključka da je kod cirkularne liste zapravo najkorisnije da koristimo vanjski pokazivač koji pokazuje na zadnji čvor, jer su tada navedene operacije najefikasnije. Zato se u nastavku za pristup pojedinim čvorovima cirkularne liste koristi pokazivač *kraj*.

Postupak umetanja novog čvora na kraj cirkularne liste je ilustriran na slici 4.21. Lista u početnom stanju sadrži brojeve 5, 12 i 8. U listu se umeće novi čvor sa sadržajem x=9. Prikazani su sljedeći koraci (slika 4.21):

- Alocira se prostor za novi čvor (slika 4.21*a*).
- U informaciono polje *info* novog čvora upisujemo sadržaj x=9 (slika 4.21*b*).
- U polje *sljedeci* novog čvora upisujemo pokazivač na prvi čvor, te na taj način povezujemo novi čvor sa prvim čvorom (slika 4.21*c*).
- U polje *sljedeci* zadnjeg čvora, na koji pokazuje *kraj*, upisujemo pokazivač na novi čvor, te na taj način povezujemo zadnji čvor sa novim čvorom (slika 4.21*d*).
- Ažuriramo vrijednost pokazivača *kraj*, tako da pokazuje na novi čvor (slika 4.21*e*).



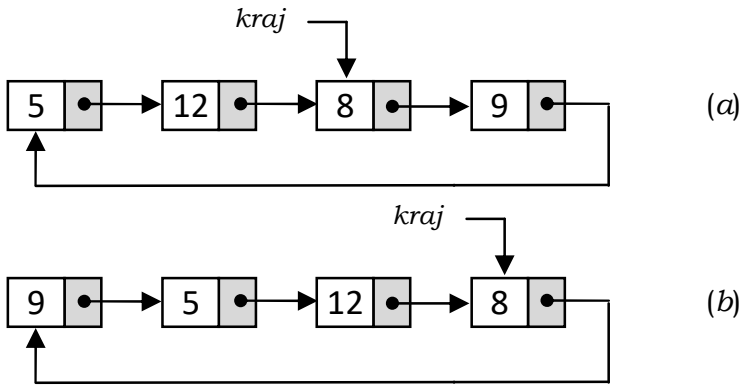
Slika 4.21. Umetanje čvora na kraj cirkularne liste

U tabeli 4.22 je prikazana funkcija CL-DODAJ-NA-KRAJ, koja opisuje umetanje novog čvora, sa informacionim sadržajem *x*, na kraj cirkularne liste *L*.

Tabela 4.22

```
    CL-DODAJ-NA-KRAJ (L, x)
1   p = GETNODE ()
2   p.info = x
3   if (L.kraj == NIL) then
4      L.kraj = p
5   else
6      p.sljedeci = L.kraj.sljedeci
7   end_if
8   L.kraj.sljedeci = p
9   L.kraj = L.kraj.sljedeci
```

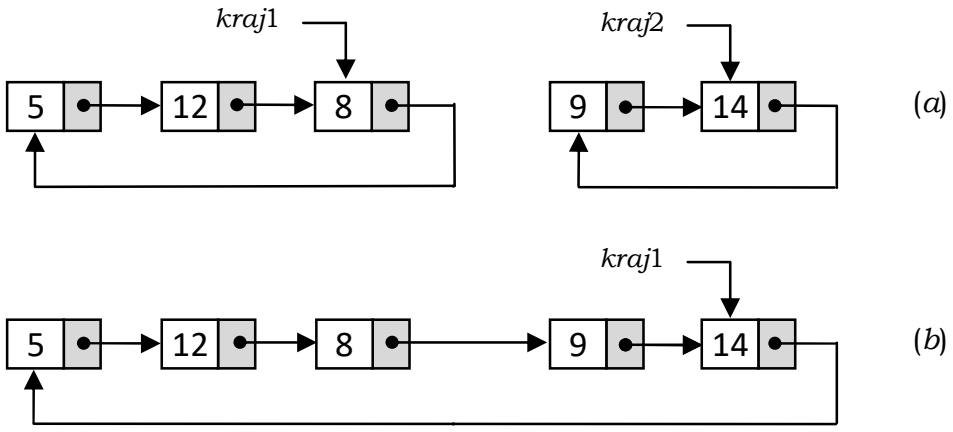
Postupak umetanja novog čvora na početak cirkularne liste je isti kao i postupak umetanja na kraj cirkularne liste, osim što nije potrebno izvršiti zadnji korak, kojim se pokazivač *kraj* ažurira tako da pokazuje na novi čvor. Prema tome, slika 4.21 ujedno ilustrira i proces umetanja novog čvora na početak cirkularne liste, pri čemu je potrebno isključiti sliku 4.21*e*, pa je zadnji korak zapravo ilustriran slikom 4.21*d*. Ovaj korak je još jednom prikazan na slici 4.22*a*, dok je na slici 4.22*b* promijenjen samo grafički prikaz liste, da bi se istaknulo da je novi čvor koji sadrži broj 9, zapravo dodan na početak cirkularne liste.



Slika 4.22. Umetanje novog čvora na početak cirkularne liste, *a*) cirkularna lista nakon umetanja novog čvora, *b*) drugačiji prikaz iste liste u kojem je istaknuto umetanje novog čvora na početak liste

Prema tome, funkcija CL-DODAJ-NA-POCETAK, koja opisuje umetanje novog čvora sa informacionim sadržajem *x* na početak cirkularne liste *L*, je ista kao i funkcija CL-DODAJ-NA-KRAJ, s jedinom razlikom da je naredbu *L.kraj = L.kraj.sljedeci*, koja se nalazi u liniji 9, potrebno izostaviti.

Postoje i neke druge operacije koje su efikasnije kada se u radu sa cirkularnom listom koristi vanjski pokazivač na zadnji čvor (*kraj*), umjesto pokazivača na prvi čvor (*pocetak*). Na slici 4.23 je prikazana operacija spajanja dvije cirkularne liste.



Slika 4.23. Spajanje cirkularnih listi

Operacija spajanja dvije cirkularne liste je opisana funkcijom CL-SPOJI u tabeli 4.23. Pošto pokazivač *L*1.*kraj* pokazuje na zadnji čvor u prvoj listi, nije potreban prolazak do kraja prve liste, da bi se spojile dvije liste. Na kraju operacije spajanja, pokazivač *L*1.*kraj* pokazuje na zadnji čvor rezultirajuće objedinjene liste.

Tabela 4.23

```
    CL-SPOJI (L1, L2)
1   if (L2.kraj ≠ NIL) then
2      if (L1.kraj ≠ NIL) then
3         p = L1.kraj.sljedeci
4         L1.kraj.sljedeci = L2.kraj.sljedeci
5         L2.kraj.sljedeci = p
6      end_if
7      L1.kraj = L2.kraj
8   end_if
9   return L1.kraj
```

4.6 Pitanja i zadaci za vježbu

1. Usporedite efikasnost tipičnih operacija kod povezanih listi, u odnosu na analogne operacije kod sekvencijalne implementacije liste (liste implementirane pomoću nizova).
2. Obrazložite koje su prednosti, te koji su nedostaci kod implementacije povezane liste, koja koristi fiktivne čvorove.
3. Obrazložite prednosti i nedostatke dvostruko povezanih listi, u odnosu na jednostruko povezane liste.
4. Razmislite u kojim situacijama i pri rješavanju kojih problema bi imalo smisla koristiti više tekućih elemenata, odnosno više pokazivača pomoću kojih se pristupa tekućim elementima.
5. Implementirajte u jeziku C++ (ili u nekom drugom programskom jeziku po vašem izboru) dvostruko povezanu listu, bez fiktivnih čvorova. Implementacija treba uključivati operacije kao i prikazani primjer implementacije s fiktivnim čvorovima u odjeljku 4.4.1.
6. Implementirajte u jeziku C++ (ili u nekom drugom programskom jeziku po vašem izboru) dvostruko povezanu cirkularnu listu, bez fiktivnih čvorova. Implementacija treba uključivati operacije definirane apstraktnom klasom `Lista`.
7. U implementaciji jednostruko povezane liste dodajte metodu `ObrniPoredakMdoN` koja uzima dva cjelobrojna argumenta *m* i *n*. Metoda treba obrnuti poredak elemenata u dijelu liste, koji je definiran brojevima *m* i *n*. Na primjer, poziv metode `ObrniPoredakMdoN(4,13)` treba obrnuti poredak elemenata koji se nalaze na pozicijama u rasponu 4..13.
8. U implementaciji povezane liste, koja je opisana u odjeljku 4.2.7, dodajte metode `NadjiMin` i `NadjiMax` pomoću kojih se pronalaze minimalni, odnosno maksimalni element u listi. Obe metode trebaju vratiti indekse pozicija pronađenih elemenata, pri čemu, u slučaju višestrukog pojavljivanja tih elemenata na različitim pozicijama, treba vratiti manji indeks.
9. Analizirajte i odredite efikasnost u *big-O* notaciji implementiranih metoda u zadatku 8.
10. U implementaciji povezane liste, koja je opisana u odjeljku 4.2.7, dodajte metode `IzbaciMin` i `IzbaciMax`, pomoću kojih se iz liste izbacuju minimalni, odnosno maksimalni element. Obe metode trebaju vratiti pronađene elemente. U rješenju možete koristiti i metode implementirane u zadatku 8.
11. Analizirajte i odredite efikasnost u *big-O* notaciji implementiranih metoda u zadatku 10.
12. U implementaciji povezane liste, koja je opisana u odjeljku 4.2.7, dodajte metodu `IzbaciMdoN`, koja kao argumente uzima cijele brojeve *m* i *n*, te iz liste izbacuje sve elemente koji se nalaze na pozicijama *m*..*n* (treba pretpostaviti da je indeks *i* prvog elementa u listi *i*=0).

13. Analizirajte i odredite efikasnost u *big-O* notaciji implementirane metode u zadatku 12.

14. U implementaciji povezane liste, koja je prikazana u odjeljku 4.2.7, dodajte metodu `ZamjeniParticije`, koja sve elemente iz lijeve particije prebacuje u desnu particiju, dok elemente iz desne particije prebacuje u lijevu particiju. Na primjer, za početnu konfiguraciju liste (8, 5, 11, 3 | 12, 7, 1, 15, 6, 14, 2), metoda `ZamjeniParticije` treba kreirati sljedeću konfiguraciju povezane liste (12, 7, 1, 15, 6, 14, 2 | 8, 5, 11, 3).

15. Analizirajte i odredite efikasnost u *big-O* notaciji implementirane metode u zadatku 14.

16. Napravite jednostavan program za rezerviranje avionskih karti. Program bi trebao prikazati meni sa sljedećim opcijama: rezerviraj kartu, otkaži rezervaciju, provjeri da li je karta rezervirana za određenu osobu i prikaži putnike. Informacije se trebaju držati u sortiranoj povezanoj listi po prezimenima. Potrebno je pretpostaviti da ne postoji ograničenje na broj letova. Kreirajte povezanu listu letova, pri čemu svaki čvor uključuje pokazivač na povezanu listu putnika.

17. Napravite jednostavan linijski editor teksta, pri čemu se čitav tekst treba držati u povezanoj listi. Nadalje, u jednom čvoru liste se treba držati tekst koji pripada jednoj liniji. Program se treba pokretati naredbom `EDIT <imedatoteke>`, nakon čega se pojavljuje prompt sa brojem linije. Program treba imati sljedeće komande za manipulaciju:

- L n	pomjeranje kursora u liniju n;
- C n_1 n_2	kopiranje linija n_1 do n_2 u privremenu listu;
- C n	kopiranje linije n u privremenu listu koja sadrži samo jedan čvor sa sadržajem iz linije n;
- P	kopiranje linije iz privremene liste na tekuću poziciju;
- X n	brisanje linije n;
- X n_1 n_2	brisanje linija n_1 do n_2;
- A	dodavanje teksta u čvor na kraj liste, koji će se unositi nakon izbora naredbe;
- E	izlazak iz programa i spašavanje datoteke na disk.

Na primjer, sljedećom sekvencom unosa sa tastature

```
EDIT tekst.txt
```

```
1>  Prvi red teksta
2>  drugi red
3>  treci red
4>  C 2 3
5>  L 1
6>  P
7>  E
```

se treba kreirati datoteka `tekst.txt` sa sljedećim sadržajem:

```
drugi red
treci red
Prvi red teksta
drugi red
treci red
```

18. Pomoću povezane liste implementirajte cjelobrojnu (integer) varijablu neograničene veličine. Svaki čvor u listi treba sadržavati jednu znamenku broja. Trebate implementirati sljedeće operacije: sabiranje, oduzimanje i množenje. Analizirajte i utvrdite kolika je vremenska složenost pojedinih operacija, pri čemu pod veličinom ulaza trebate podrazumijevati ukupan broj znamenki, koje se koriste u operaciji.