

RI301

Strukture podataka

dr.sc. Edin Pjanić

Pregled predavanja

- Lista kao apstraktni tip podatka (ATP)
- Implementacija ATP liste pomoću niza sukcesivnih memorijskih lokacija (nizom)
- Implementacija ATP liste pomoću povezanih memorijskih lokacija (jednostruko povezanih)

Apstraktni tip podataka (ATP)

- Engl.: Abstract data type (ADT).
- Apstraktni tip podataka:
 - Tip podataka kod kojeg su njegova logička svojstva odvojena od detalja implementacije.
- Tri karakteristike ATP:
 - Ime tipa,
 - Domena (skup vrijednosti koje pripadaju tom tipu) i
 - Skup operacija nad podacima.
- Apstraktni tip podataka ne definiše konkretnu implementaciju (enkapsulacija).
- U implementaciji se mora osmisliti kako predstaviti podatke i napisati algoritme za definisane operacije.

Lista (ATP)

- Kolekcija **elemenata**, obično istog tipa, sa definisanim **poretkom** (pozicijom svakog elementa).

- Primjeri:

Lista za kupovinu

mljeko
hljeb
jaja
voda
kafa
čokolada
neka igračkica

Bingo brojevi

27	73
64	82
89	90
23	6
5	21
63	56
79	25
28	48
45	87
8	4
66	49
20	37
22	54
55	39
57	24
9	62
16	46
32	53
81	51
60	38
35	50
59	83
11	5

Džeko
Ibišević
Misimović
Bolić
Barbarez
Baljić
Pjanić
Muslimović
Medunjanin
Đurić

Lista strijelaca
fudbalske
reprezentacije BiH

Lista komandi

- Get L
- Forward L
- Turn left
- Forward L/2
- Dig 30 cm
- ...

Lista (ATP)

- Apstraktni prikaz liste:

$\langle a_0, a_1, a_2, a_3, \dots, a_{n-1} \rangle$ ili $(a_0, a_1, a_2, a_3, \dots, a_{n-1})$

ili $\langle a_1, a_2, a_3, a_4, \dots, a_n \rangle$ ili $(a_1, a_2, a_3, a_4, \dots, a_n)$

- Još neki pojmovi:
 - **Prazna lista (*empty*)**: lista nema elemenata $\langle \rangle$.
 - **Dužina ili veličina (*length*)**: broj elemenata u listi.
 - **Glava (*head*)**: prednji kraj liste (prvi element).
 - **Rep (*tail*)**: stražnji kraj liste (posljednji element).

Lista (ATP)

- Najčešće operacije koje se definišu nad listom kao ATP:
 - kreiranje liste,
 - dodavanje elementa u listu,
 - uklanjanje elementa iz liste,
 - dobijanje vrijednosti određenog elementa u listi,
 - dobijanje veličine liste itd.
- Pri implemetaciji operacija vrlo važan koncept je **trenutna pozicija**, koja označava mjesto gdje će se obaviti neka operacija (dodavanje, uklanjanje, ...).
- Ovaj koncept se implementira:
 - Internim stanjem (indeks, pokazivač i sl.).
 - Posebnim objektom koji predstavlja poziciju (iterator).

Lista (ATP) – kretanje po listi

Elementi liste: $\langle a_1, a_2, a_3, a_4, \dots, a_{n-1}, a_n \rangle$

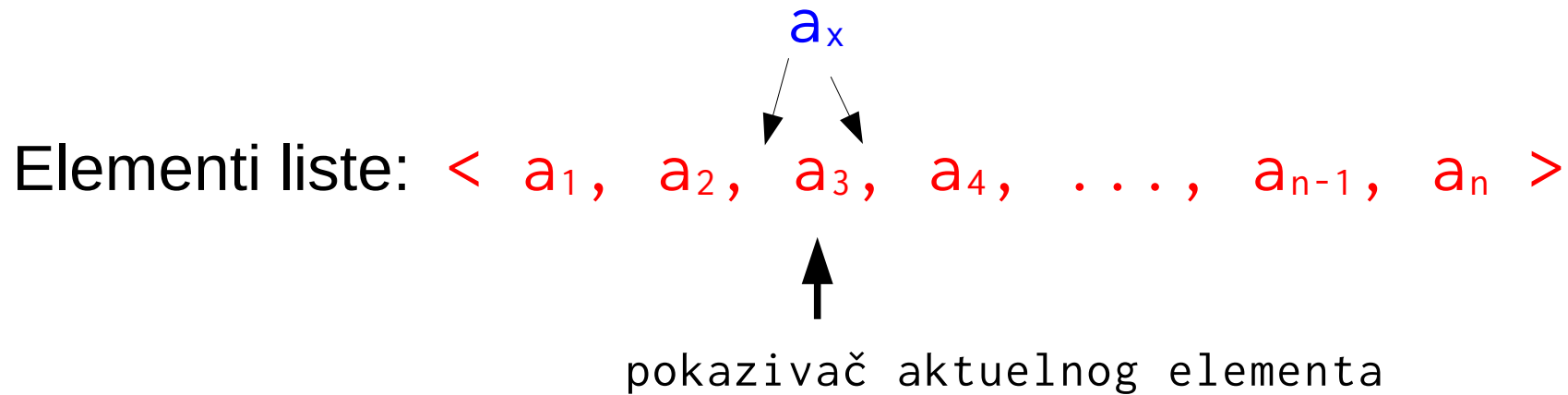


pokazivač aktuelnog elementa

Neke operacije sa aktuelnim (trenutnim) elementom:

- čitanje/mijenjanje vrijednosti
- brisanje elementa
- dodavanje novog elementa na tu poziciju

Lista (ATP) – dodavanje novog elementa



Dodavati možemo:

- iza trenutnog elementa $\langle a_1, a_2, a_3, a_x, a_4, \dots, a_{n-1}, a_n \rangle$
- ispred trenutnog elementa $\langle a_1, a_2, a_x, a_3, a_4, \dots, a_{n-1}, a_n \rangle$

$\underbrace{\hspace{15em}}$
n+1 element

Lista (ATP) – primjer uklanjanja elementa

Elementi liste: < 5, 8 -7, 22, 53, 19 >



pokazivač aktuelnog elementa

Nakon uklanjanja 'trenutnog' elementa:

Elementi liste: < 5, 8 -7, 53, 19 >



pokazivač aktuelnog elementa

Lista – implementacija operacija

- Operacije nad strukturom podataka ćemo implementirati unutar metoda ili funkcija.
- Na osnovu prethodnog razmatranja mogli bismo navesti nazive nekih metoda/funkcija koje će imati naša Lista bez obzira na implementaciju:
 - dodaj, dodaj_ispred, dodaj_iza, insert, ...
 - ukloni, erase, ...
 - trenutni, current, *, ...
 - naprijed, forward, ++
 - nazad, backward, --
 - velicina, size
 - itd.

} nad iteratorom

Lista – specifikacija

Metode koje konkretna implementacija treba implementirati možemo specificirati u obliku:

```
template<typename TipElementa>
class Lista { //
    public:
        ~Lista();
        bool dodaj(iterator poz, const TipElementa& novi);
        bool dodajNaKraj(const TipElementa& novi);
        void ukloni();
        void isprazni();
        TipElementa& trenutniElement();
        void naPocetak();
        void naKraj();
        void naPrethodni();
        void naNaredni();
        void uradiNesto();
        size_t velicina();
};
```

Lista – specifikacija

Metode možemo predvidjeti i na engleskom jeziku (poželjno):

```
template<typename ElemType>
class List { // Klasa List kao ATP
public:
    ~List();
    bool insert(iterator pos, const ElemType& val);
    bool push_back(const ElemType& val);
    bool push_front(const ElemType& val);
    ElemType pop_back(); // Ili void
    ElemType pop_front(); // Ili void
    ElemType & back();
    ElemType & front();
    void remove(const ElemType& val);
    void clear();
    size_t size();
    iterator begin();
    iterator end();
};
```

Primjer rada sa listom: *std::list*

```
#include <list>
#include <iostream>

void repeat_back(std::list<int>& numbers, size_t count)
{
    auto num = numbers.back();
    for(size_t i=0; i<count; ++i){
        numbers.push_back(num);
    }
}
```

Funkcija koja će zadani broj puta klonirati zadnji element u listi.

```
void repeat_front(std::list<int>& numbers, size_t count)
{
    auto num = numbers.front();
    for(size_t i=0; i<count; ++i){
        numbers.push_front(num);
    }
}
```

Funkcija koja će zadani broj puta klonirati prvi element u listi.

```
template<typename T>
void print(const std::list<T>& l, std::ostream& out)
{
    for(auto it=l.begin(); it!=l.end(); ++it)
        out << *it << ' ';
}
```

Funkcija koja će ispisati na standardni izlaz sve elemente u listi.

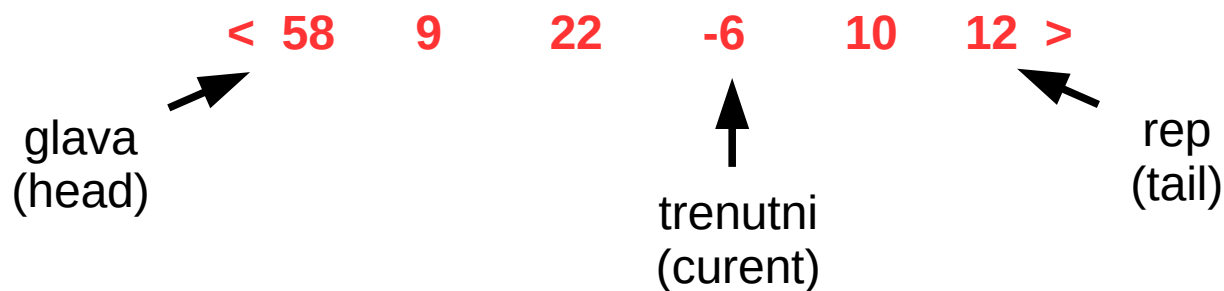
Lista (ATP)

- Neki načini implementacije:
 - nizom (array)
 - povezanim memorijskim lokacijama (linked memory)
- Performanse operacija zavise od načina implementacije.
- Posebni oblici ove strukture podataka su:
 - stack (stog),
 - queue (red).

Lista – implementacija pomoću niza

- Elemente liste možemo smjestiti u niz, obično dinamički alociran. Primjer:

Lista kao ATP:



Implementacija liste pomoću niza:

trenutni (indeks): 3

veličina (liste): 6

glava: 0

rep: 5

kapacitet: 9

58	9	22	-6	10	12	?	?
0	1	2	3	4	5	6	7
(1	2	3	4	5	6)		

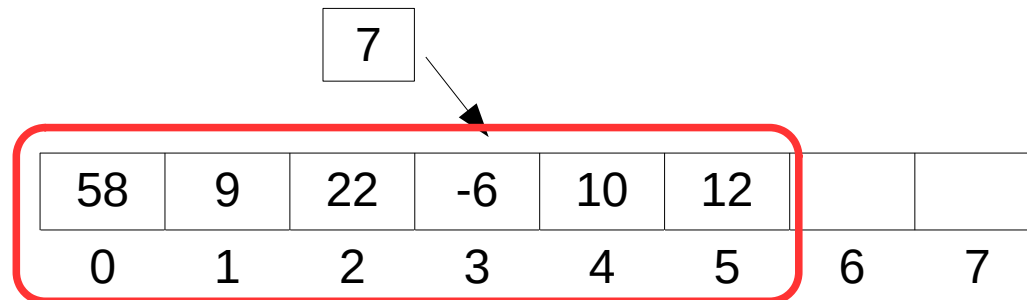
dio niza koji se koristi za listu

Ovdje ne smijemo poistovjećivati listu i niz.
Niz se koristi kao prostor za smještanje elemenata liste.

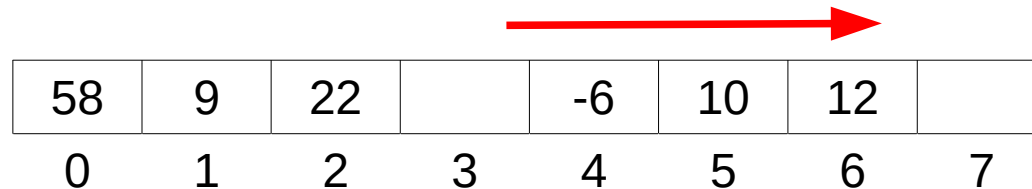
Lista – dodavanje elementa

- Novi element u listu možemo dodati ispred ili iza trenutne pozicije. Primjer dodavanja ispred:

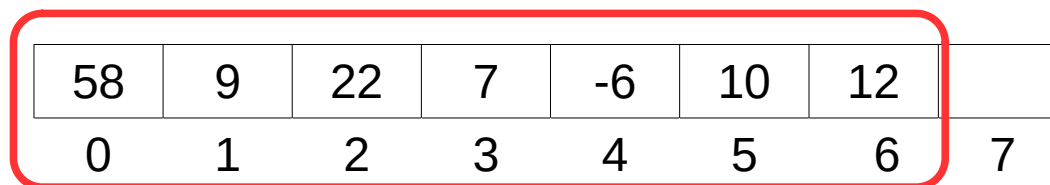
trenutni (indeks): 3
veličina: 6



premjestiti



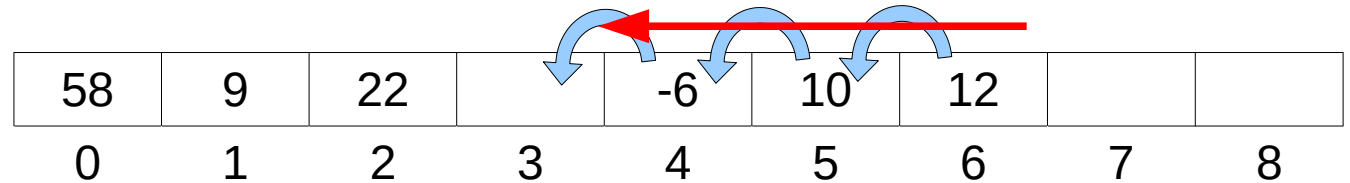
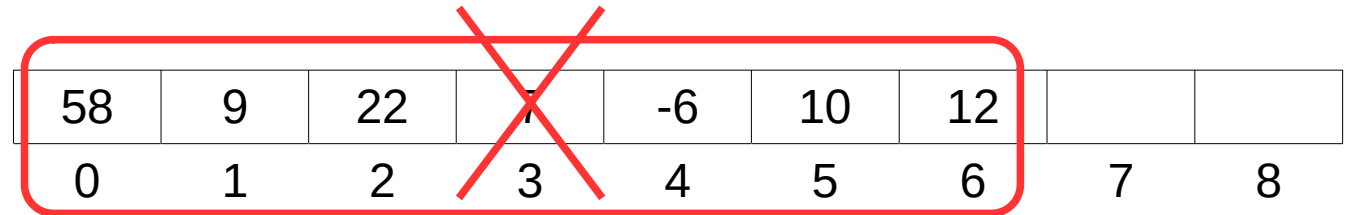
trenutni (indeks): 3
veličina: 7



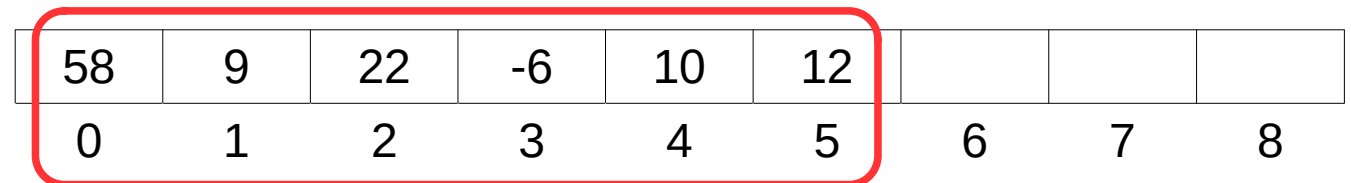
Složenost: $O(n)$

Lista – uklanjanje elementa

trenutni (indeks): 3
velicina: 7



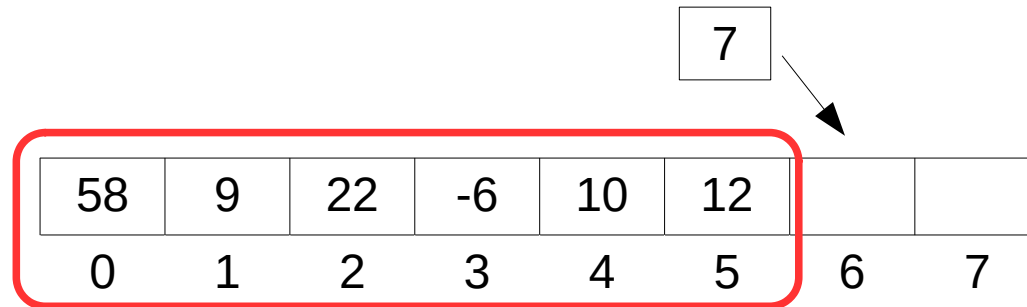
trenutni (indeks): 3
velicina: 6



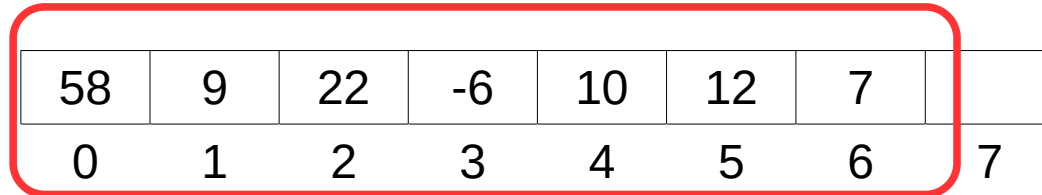
Složenost: $O(n)$

Lista – dodavanje elementa na kraj liste

trenutni (indeks): 3
veličina: 6



trenutni (indeks): 3
veličina: 7



Složenost: $O(1)$

Lista – uklanjanje elementa sa kraja liste

trenutni (indeks): 3

veličina: 7

58	9	22	-6	10	12	7	
0	1	2	3	4	5	6	7

trenutni (indeks): 3

veličina: 6

58	9	22	-6	10	12		
0	1	2	3	4	5	6	7

Koja je složenost uklanjanja/dodavanja na početak liste?

Složenost: $O(1)$

Lista nizom – složenost nekih operacija

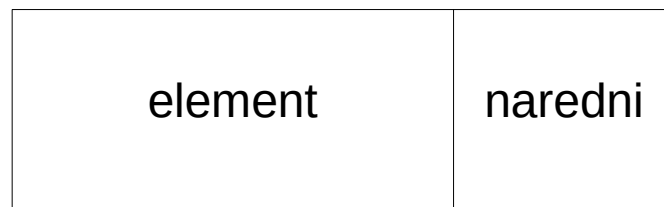
- Pristup trenutnom elementu: $O(1)$
- Dodavanje novog elementa (na proizvoljnu lokaciju): $O(n)$
- Uklanjanje elementa (sa proizvoljne lokacije): $O(n)$
- Dodavanje novog elementa na kraj liste: $O(1)$
- Uklanjanje elementa sa kraja liste: $O(1)$
- Dodavanje novog elementa na početak liste: $O(n)$
- Uklanjanje elementa sa početka liste: $O(n)$
- Traženje elementa po vrijednosti: $O(n)$

- Ostale operacije?
- Iterator, tip iteratora...?

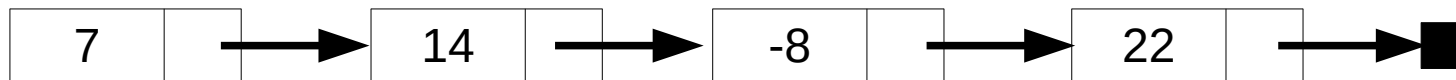
58	9	22	-6	10	12		
0	1	2	3	4	5	6	7

Lista (ATP) – implementacija pomoću povezanih memorijskih lokacija

- Ideja je da ubrzamo dodavanje i uklanjanje elemenata tako što nema pomjeranja ostalih elemenata.
- Osnova svega je čvor (član, atom) koji sadrži **element liste** i **vezu** na naredni element.



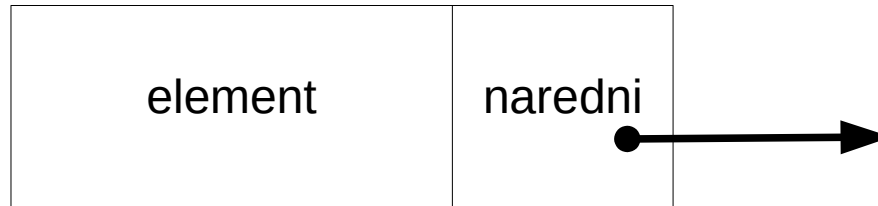
- Listu dobijemo uvezivanjem čvorova na slj. način:



- Lista: (7, 14, -8, 22)

Čvor liste

```
struct Cvor
{
    int element;
    Cvor * naredni;
    Cvor(int e) : element(e), naredni(nullptr) {}
};
```

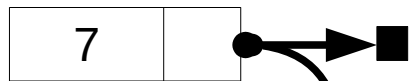


```
template <typename Elem>
class Cvor
{
public:
    Elem element;
    Cvor* naredni;
    Cvor(const Elem & e) : element(e), naredni(nullptr) {}
};
```

Uvezivanje čvorova liste – osnovni principi

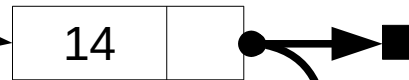
Lista: < 7, 14, -8, 22 >

```
Cvor * a = new Cvor(7);
```



```
Cvor * b = new Cvor(14);
```

```
(*a).naredni = b;  
a->naredni = b;
```



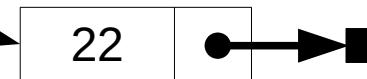
```
Cvor * c = new Cvor(-8);
```

```
b->naredni = c;
```



```
c->naredni = d;
```

```
Cvor * d = new Cvor(22);
```



```
struct Cvor  
{  
    int element;  
    Cvor* naredni;  
    Cvor(int e)  
        : element(e), naredni(nullptr)  
    {}  
};
```

Svaki čvor se dinamički alocira.

Zadnji čvor pokazuje na null-pointer što nam je indikacija da nema više čvorova.

U nekim implementacijama zadnji čvor pokazuje na samog sebe. U nekima na specijalni čvor koji označava kraj liste.

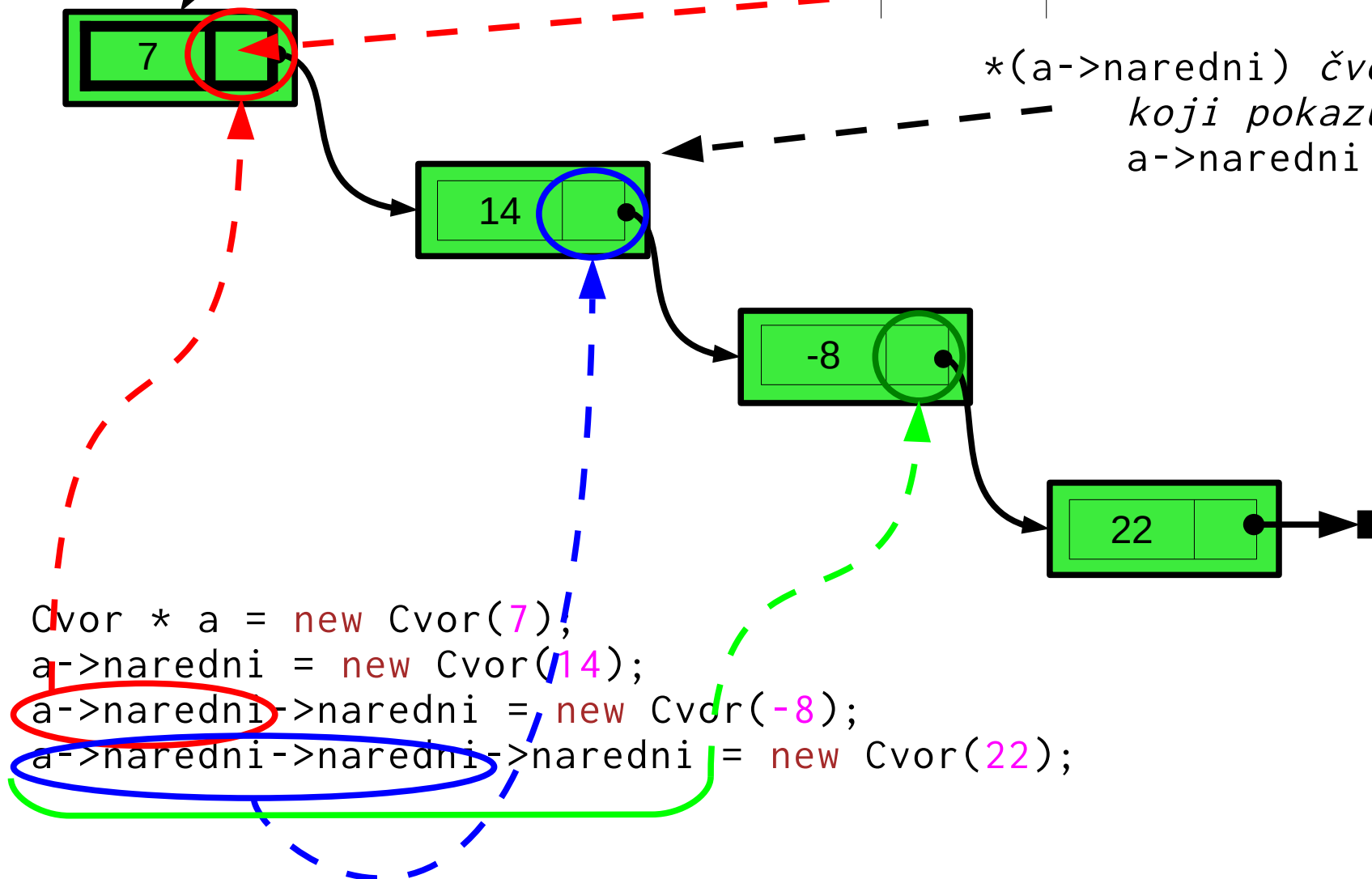
Uvezivanje čvorova liste

Lista: < 7, 14, -8, 22 >

Cvor * a

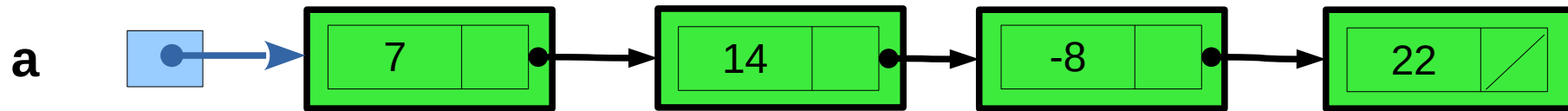
a->naredni je pokazivač

*(a->naredni) čvor na koji pokazuje a->naredni



Kretanje po čvorovima liste

Lista: < 7, 14, -8, 22 >



```
void ispisi(Cvor *c)
{
    while(c != nullptr)
    {
        std::cout << c->element << " ";
        c = c->naredni;
    }
    std::cout << " KRAJ" << std::endl;
}
```

ispisi(a);

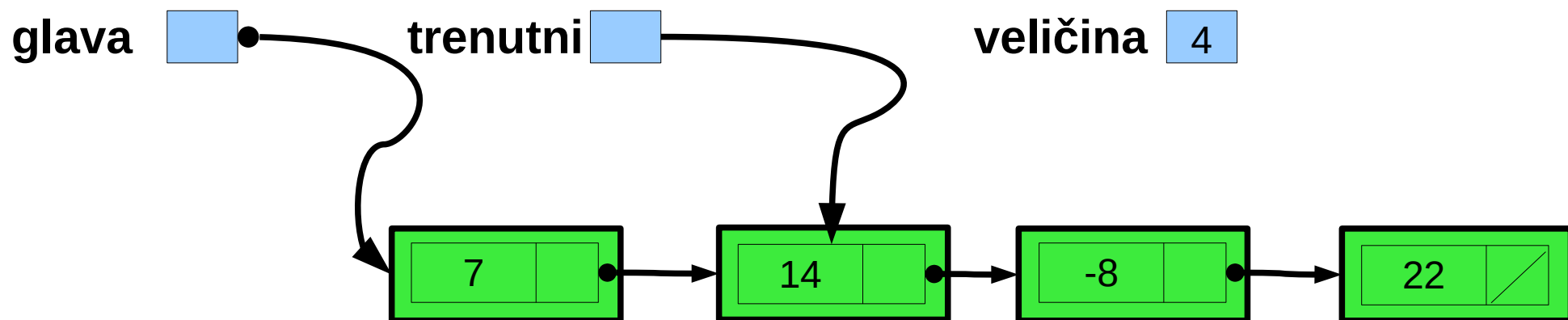
=> 7 14 -8 22 KRAJ

```
ispisi(a);
ispisi(a->naredni);
ispisi(a->naredni->naredni->naredni);
```



7	14	-8	22	KRAJ
14	-8	22	KRAJ	
22	KRAJ			

Lista – glavni elementi



- **glava** – pokazivač na početak liste (prvi čvor). Ovaj pokazivač ne smijemo izgubiti inače bismo izgubili cijelu listu.
- **trenutni** – pokazivač na trenutni element u listi koji je referenca za dodavanje ili uklanjanje čvora i sl. Obično je to eksterni objekat - iterator.
- **veličina** – broj čvorova liste.

Lista – implementacija (int)

```
class Lista
{
    private:
        class Cvor
        {
            public:
                int element;
                Cvor* naredni;
                Cvor(int v) : element(v), naredni(nullptr) {}
        };
        Cvor * glava;
        Cvor * trenutni_cvor;
        size_t velicina_liste;
    public:
        Lista();
        ~Lista();
        void naprijed();
        void nazad();
        void dodaj_iza(int X);
        void dodaj_ispred(int X);
        // ostali metodi
};
```

Samo u edukativne svrhe (privremeno).
U stvarnosti se za kretanje po listi koristi
iterator.

`void dodaj(iterator it, int X);`

U opštem slučaju (template):
`void insert(iterator it, const T& x);`

Lista – konstruktor i destruktor

Konstruktor

- Kreirati praznu listu (bez elemenata.)

Destruktor

- Svaki čvor je dinamički alociran.
- Dealocirati svu dinamički alociranu memoriju.
 - Proći kroz svaki čvor liste i dealocirati ga.

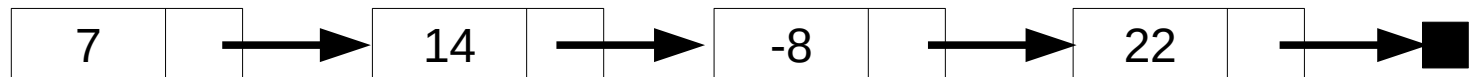
Lista – konstruktor i destruktor

```
Lista::Lista() // kreira se prazna lista
: glava(nullptr),
  trenutni_cvor(nullptr),
  velicina_liste(0)
{}
```

Složenost:
 $O(1)$

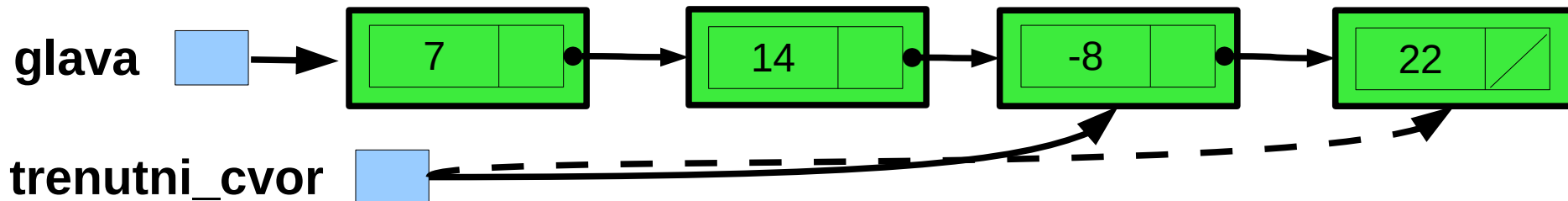
```
Lista::~~Lista()
{
    Cvor *temp;
    while(glava)
    {
        temp = glava->naredni;
        delete glava;
        glava = temp;
    }
}
```

Složenost:
 $O(n)$



Pomjerenje 'trenutnog' naprijed

Lista: < 7, 14, -8, 22 >



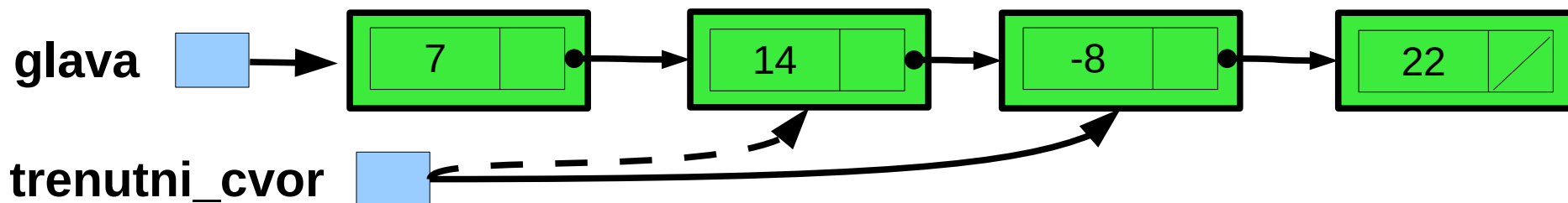
Da bismo trenutni marker (ili iterator) pomjerali naprijed potrebno je da trenutni_cvor (ili iterator) postavimo na naredni čvor onog na koji on pokazuje.

```
trenutni_cvor = trenutni_cvor->naredni;
```

Složenost: $O(1)$

Pomjerenje unazad

Lista: < 7, 14, -8, 22 >



Da bismo trenutni marker pomjerali nazad potrebno je da privatni član **trenutni_cvor** postavimo na čvor čiji član **naredni** pokazuje na trenutni čvor.

S obzirom na to da nikako ne možemo doći direktno do tog čvora jer su čvorovi uvezani samo u jednom smjeru, moramo krenuti od prvog čvora (**glava**) dok ne dođemo do traženog.

Dakle, prolazimo kroz čvorove. Kad nađemo na čvor čiji pokazivač **naredni** pokazuje na trenutni (ima istu vrijednost kao **trenutni_cvor**) to je traženi čvor.

Složenost: $O(n)$

Lista – pomjerenje naprijed i nazad


```
void Lista::naprijed()
{
    if(trenutni_cvor && trenutni_cvor->naredni)
        trenutni_cvor = trenutni_cvor->naredni;
    else
        throw out_of_range("Ne mozemo nakon zadnjeg cvora.");
}
```

$O(1)$

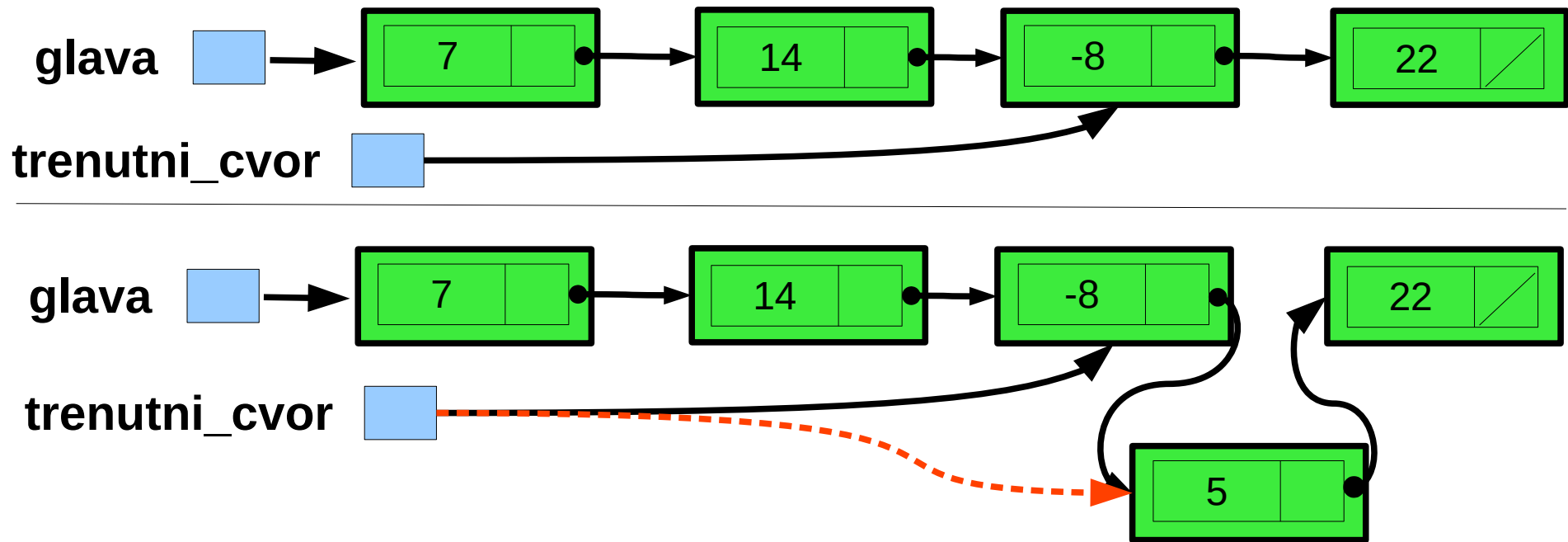
```
void Lista::nazad()
{
    if(trenutni_cvor == glava)
        throw out_of_range("Ne mozemo ispred prvog cvora.");

    Cvor *temp = glava;
    while (temp->naredni != trenutni_cvor)
        temp = temp->naredni;
    trenutni_cvor = temp;
}
```

$O(n)$

 Nemamo direktnu vezu na prethodni čvor, pa moramo krenuti od početka liste.

Dodavanje novog čvora (iza trenutnog)



Da bismo dodali novi čvor u listu potrebno je kreirati novi čvor i samo promijeniti dva pokazivača: naredni novog čvora treba podesiti da pokazuje na čvor koji je bio poslije trenutnog a naredni trenutnog čvora treba podesiti da pokazuje na novi čvor.

Treba voditi računa o redoslijedu ovih operacija da ne bismo izgubili dio liste nakon trenutnog čvora.

Složenost: $O(1)$

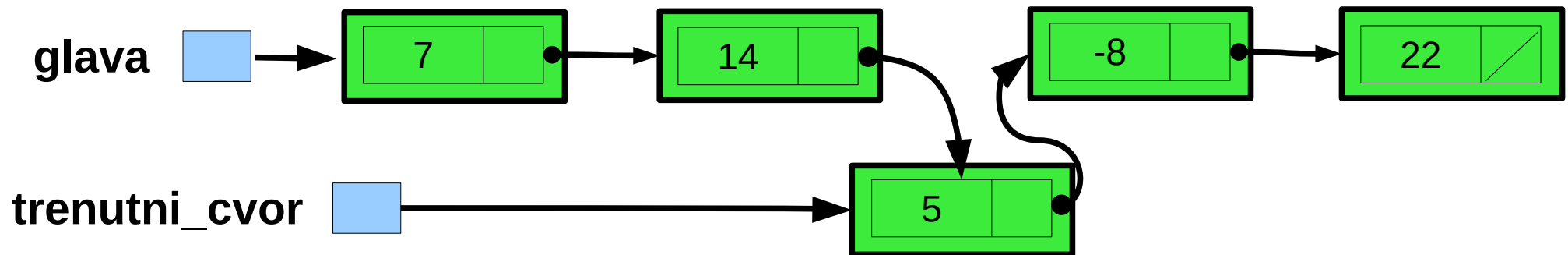
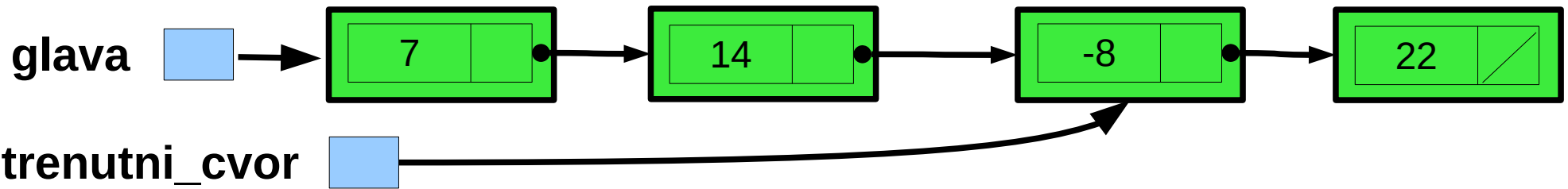
Dodavanje novog čvora (iza trenutnog)

Uvijek treba voditi računi i o posebnim slučajevima, kao što je dodavanje novog čvora u praznu listu.

```
void Lista::dodaj_iza(int X)
{
    Cvor * temp = new Cvor(X);
    if(velicina_liste != 0) // ako lista nije prazna
    {
        temp->naredni = trenutni_cvor->naredni;
        trenutni_cvor->naredni = temp;
        trenutni_cvor = temp;
    }
    else
    {
        glava = temp;
        trenutni_cvor = temp;
    }
    velicina_liste++;
}
```

$O(1)$

Dodavanje novog čvora (ispred trenutnog)



Da bismo dodali novi čvor u listu ispred trenutnog čvora, moramo nekako doći do čvora ispred trenutnog. Jedini način je da krenemo od početka liste.

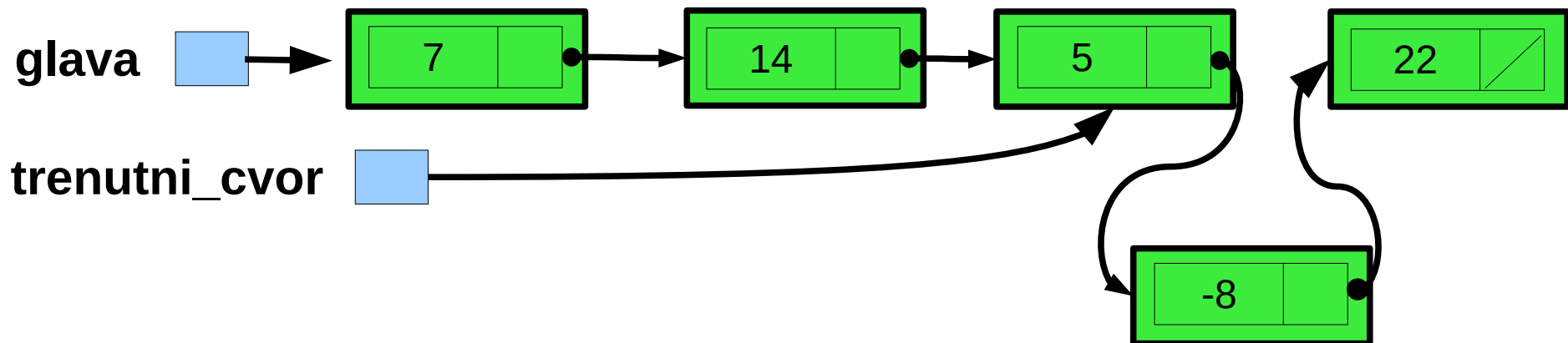
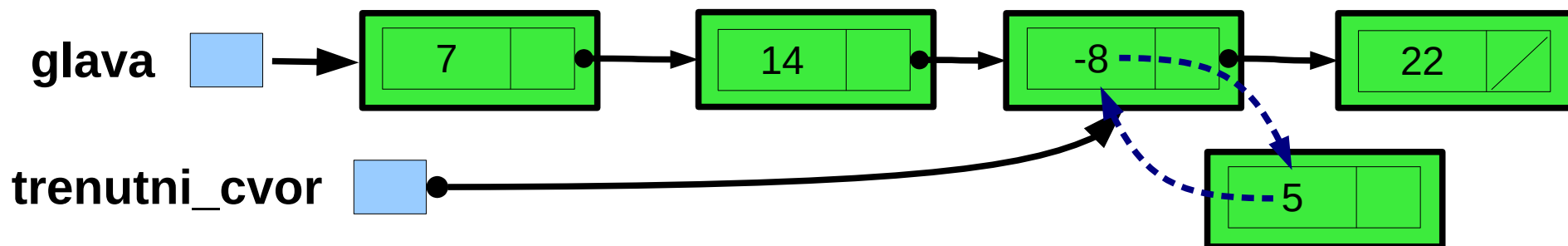
Nakon toga, manipulacija pokazivačima na naredni čvor je slična kao pri dodavanju čvora iza trenutnog.

Složenost: $O(n)$

Dodavanje novog čvora ispred trenutnog – druga verzija

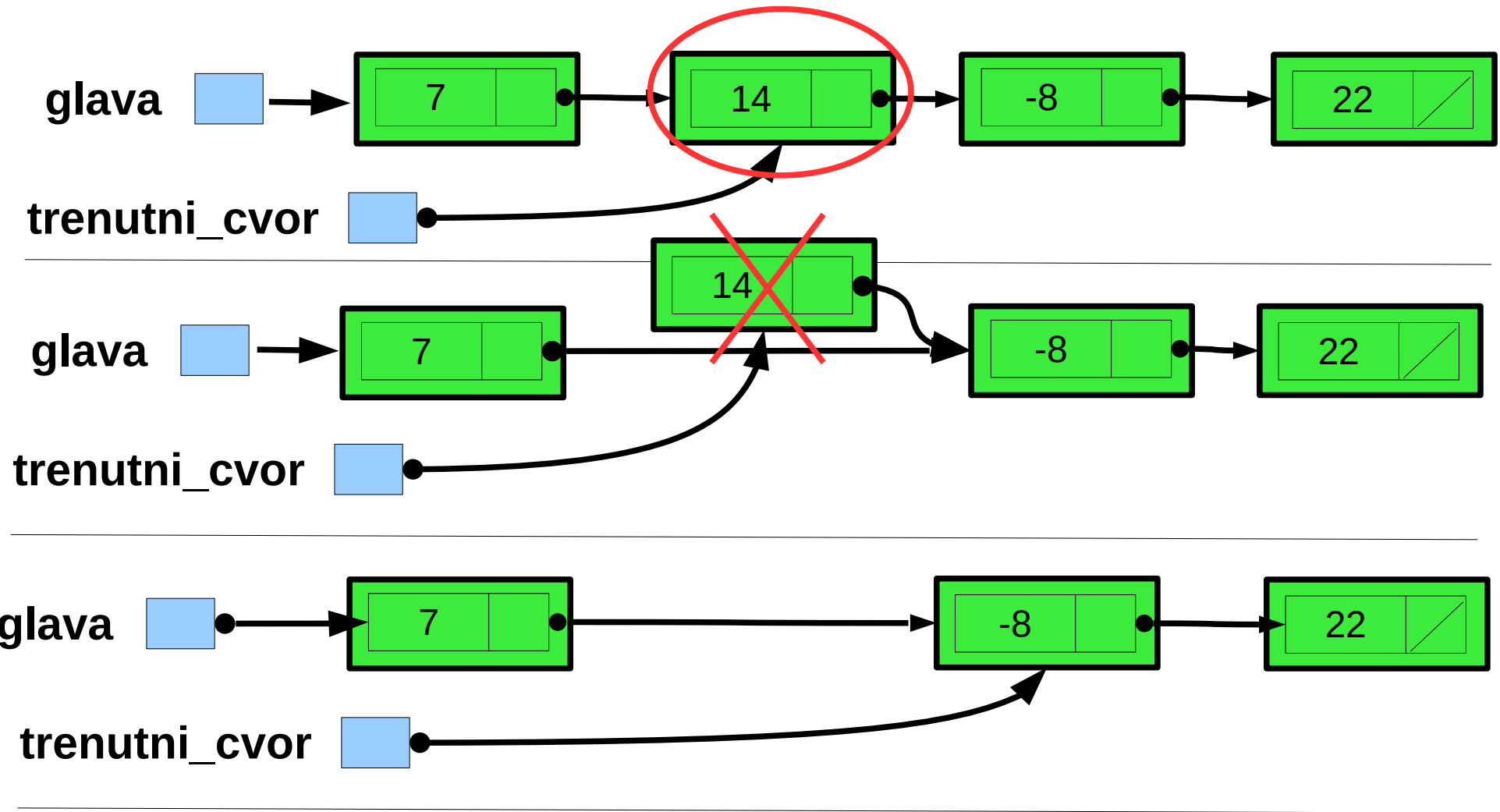
Možda ne moramo ići od početka liste?

Mogli bismo dodati čvor iza trenutnog a onda samo zamijeniti vrijednost elementa (podatka) u novom čvoru sa vrijednošću u trenutnom čvoru.



Složenost: $O(1)$

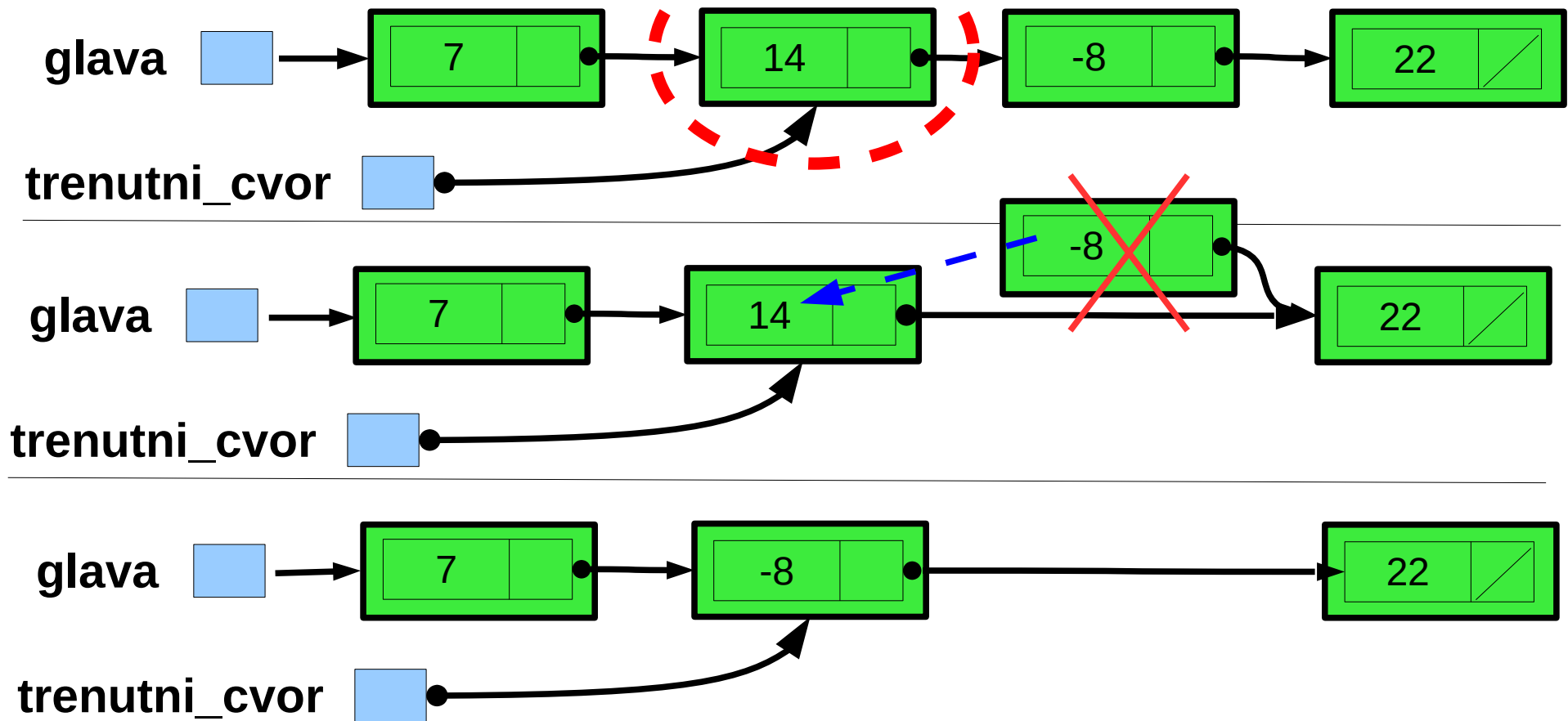
Uklanjanje čvora iz liste



Opet imamo problem sa manipulacijom čvora koji je ispred trenutnog čvora – moramo doći do njega od početka.

Složenost: $O(n)$

Uklanjanje čvora iz liste – druga verzija



Umjesto da fizički brišemo trenutni čvor, iskopiramo element (podatak) iz narednog čvora u trenutni a brišemo naredni čvor.

Podesiti pokazivače ispravno!

Imamo mali problem pri brisanju zadnjeg čvora. $O(n)$

Razmotriti i slučaj liste sa samo jednim čvorom.

Složenost: $O(1)$

Uklanjanje čvora (druga verzija)

```
void Lista::ukloni()
{
    Cvor *temp;
    if(trenutni_cvor == nullptr) return; // Ako nema cvorova
    if(trenutni_cvor->naredni){ // Ako nije zadnji cvor primijeni "trik"
        temp = trenutni_cvor->naredni;
        trenutni_cvor->element = temp->element;
        trenutni_cvor->naredni = temp->naredni;
        delete temp;
    }
    else if(velicina_liste == 1){ // Ako je samo jedan cvor izbrisi ga
        delete glava;
        glava = trenutni_cvor = nullptr;
    }
    else { // Ako je zadnji cvor, nadji njegov prethodni od pocetka
        temp = glava;
        while(temp->naredni != trenutni_cvor) temp = temp->naredni;
        temp->naredni = nullptr;
        delete trenutni_cvor;
        trenutni_cvor = temp;
    }
    --velicina_liste;
}
```

$O(1)$

$O(n)$

Pristup elementu liste po indeksu

- Ako bismo željeli implementirati pristup po indeksu korištenjem operatora `[]` na način:

```
Lista a; int e;
```

```
...
```

```
e = a[3];
```

- Da bismo došli do i -tog čvora moramo krenuti od početka liste.
- Liste rijetko imaju ovaj operator.

Složenost: $O(n)$

Operator [] - primjer implementacije

```
int Lista::operator[](size_t i)
{
    if(i >= velicina_liste)
        throw std::out_of_range("Indeks premasuje velicinu liste.");

    Cvor *temp = glava;
    while(i != 0)
    {
        temp = temp->naredni;
        --i;
    }
    return temp->element;
}
```

$O(n)$

Korisni dodaci

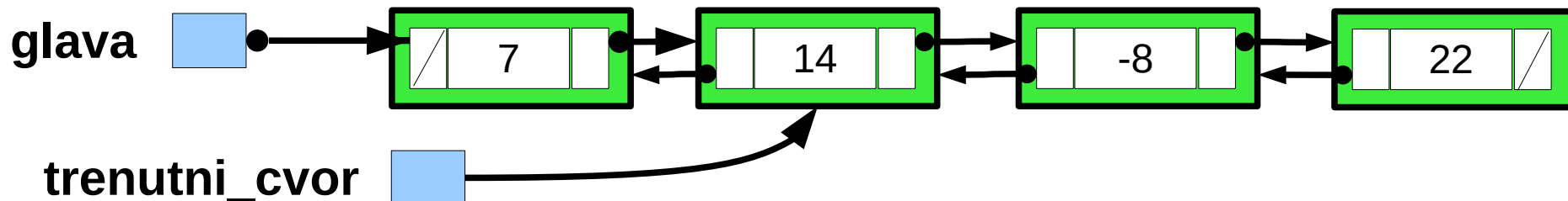
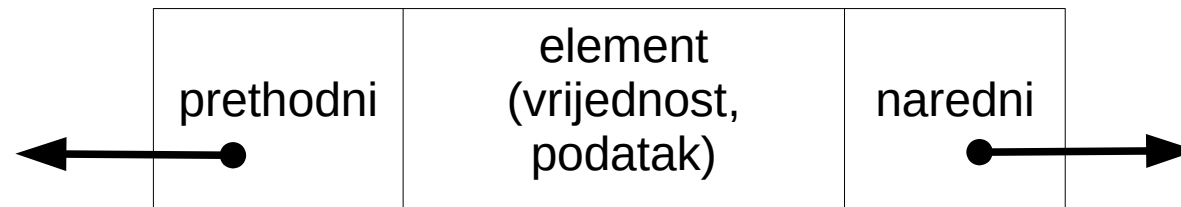
- Ponekad je korisno imati mogućnosti:
 - direktnog dodavanja novog elementa na početak ili kraj liste
 - uklanjanje elementa sa početka ili kraja liste
 - Vezu od zadnjeg ka prvom elementu (ciklična lista)
- U tu svrhu bi trebalo definisati odgovarajuće metode (dodaj_na_pocetak, ukloni_sa_kraja, ...)
- Osim toga, da bi se efikasno realizovale ove operacije (metodi) potrebno je u klasu Lista dodati pokazivač koji uvijek pokazuje na zadnji čvor liste (rep).
- Korisno je definisati iterator za korišćenje generičkih algoritama i sl.
- Metode imenovati na engleskom i što usklađenije sa standardima (push_back, clear, delete, insert, find)

Sortirana lista

- Lista koju smo do sad analizirali je nesortirana lista u smislu da čvorovi nisu poredani po nekom posebnom kriteriju.
- Kod sortirane liste čvorovi su poredani po odgovarajućem kriteriju, najčešće od "najmanjeg" do "najvišeg" elementa.
- Kod ovakve liste:
 - nema smisla imati više metoda za dodavanje čvora
 - svako dodavanje zahtijeva traženje pozicije
 - ostale operacije su manje-više identične sa nesortiranom listom
 - prilikom prolaska kroz listu uvijek imamo sortirane elemente, što je zgodno u nekim slučajevima

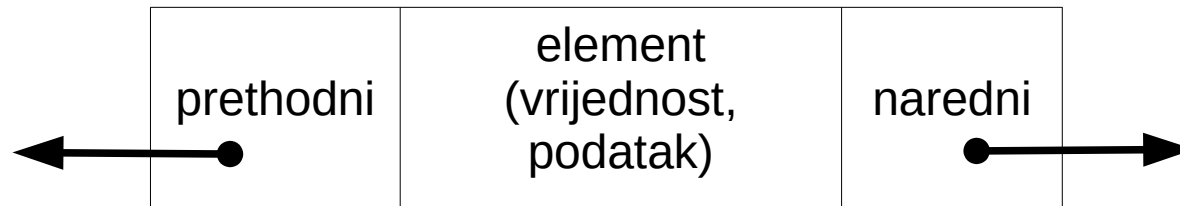
Dvostruko povezana lista

- Mane jednostruko povezane liste:
 - prolazak kroz listu samo u jednom smjeru
 - vraćanje unazad je $O(n)$ (moramo ići od početka)
- Dvostruko povezane liste u svakom čvoru imaju vezu na oba kraja, tj. i na svog prethodnika i na svog sljedbenika u listi.
STL lista `std::list` (zaglavlje `<list>`) je implementirana kao dvostruko povezana lista.



Čvor liste

```
class Cvor
{
public:
    int element;
    Cvor* prethodni;
    Cvor* naredni;
    Cvor(int e) : element(e), naredni(nullptr), prethodni(nullptr)
}
};
```



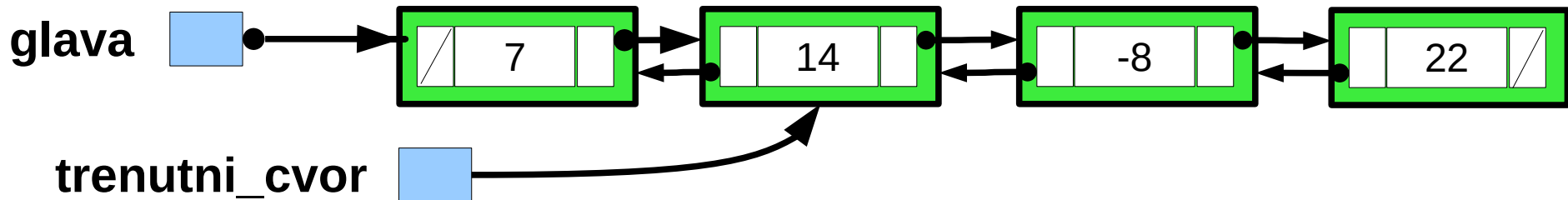
```
template <typename Elem>
class Cvor
{
public:
    Elem element;
    Cvor* prethodni;
    Cvor* naredni;
    Cvor(const Elem & e) : element(e),
                          prethodni(nullptr), naredni(nullptr) {}
};
```

Neke karakteristike dvostruko povezane liste

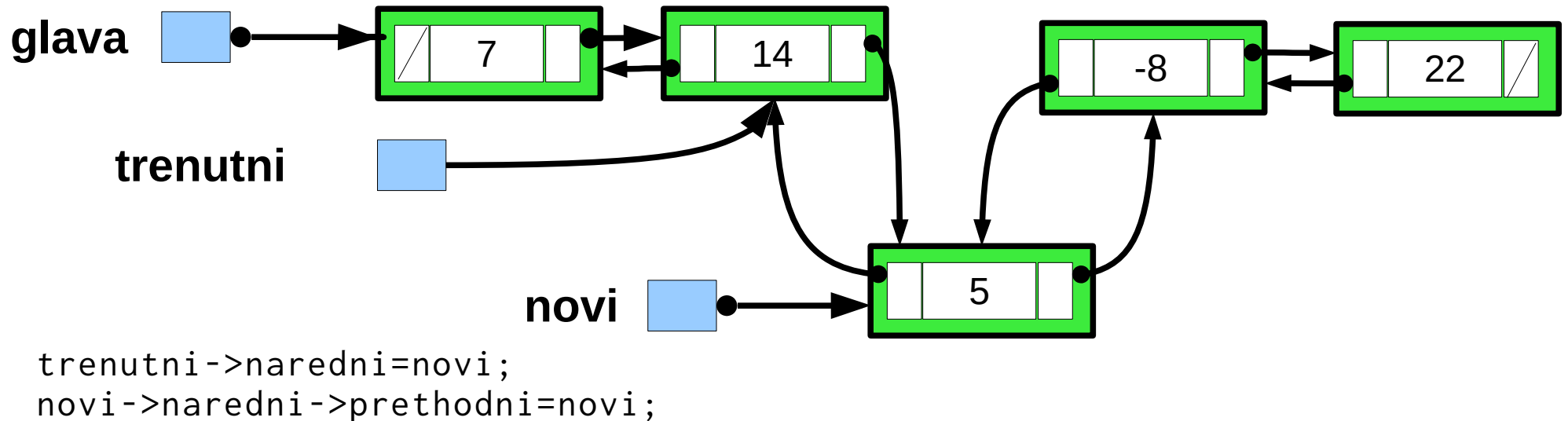
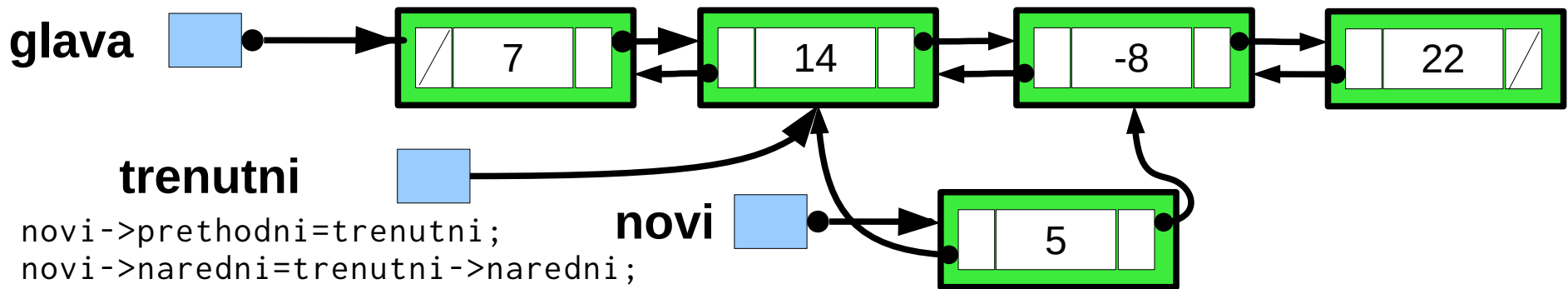
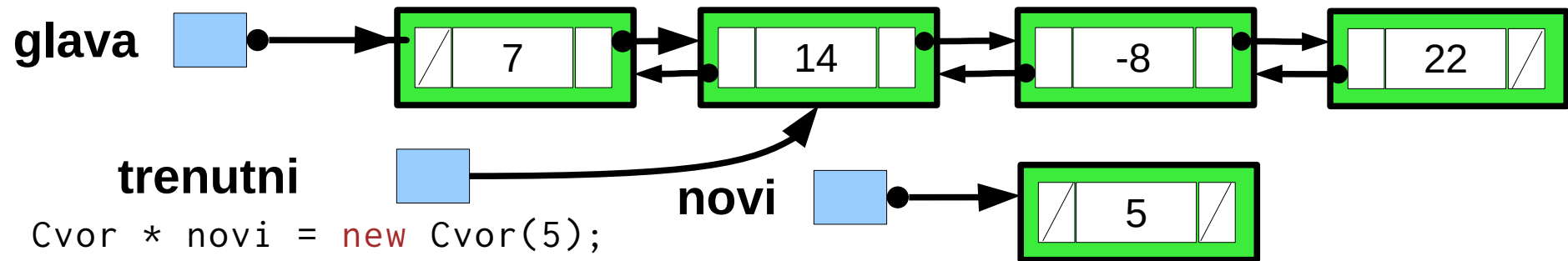
- Dodatni pokazivač zahtijeva $O(n)$ dodatne memorije, ali je to neznatno.
- Kretanje kroz listu je moguće u oba smjera na isti način $O(1)$.
- Dodavanje ispred i iza datog elementa ima složenost $O(1)$.
- Uklanjanje čvora ima složenost $O(1)$.
- Dodavanje na početak ili kraj liste (zahtijeva poseban pokazivač, npr. rep) je $O(1)$
- Uklanjanje sa početka ili kraja liste (rep) je $O(1)$
- Traženje elementa po vrijednosti i dalje ima $O(n)$ jer je potrebno proći kroz listu.

Dodavanje (uklanjanje) novog čvora (elementa)

- Dodavanje (uklanjanje) novog čvora je slično kao i kod jednostruko povezane liste. Međutim, potreban je dodatni oprez pri manipulaciji pokazivačima na naredni i prethodni čvor.
- Primjer: neka je data sljedeća lista u koju treba dodati novi element vrijednosti 5 iza trenutnog.



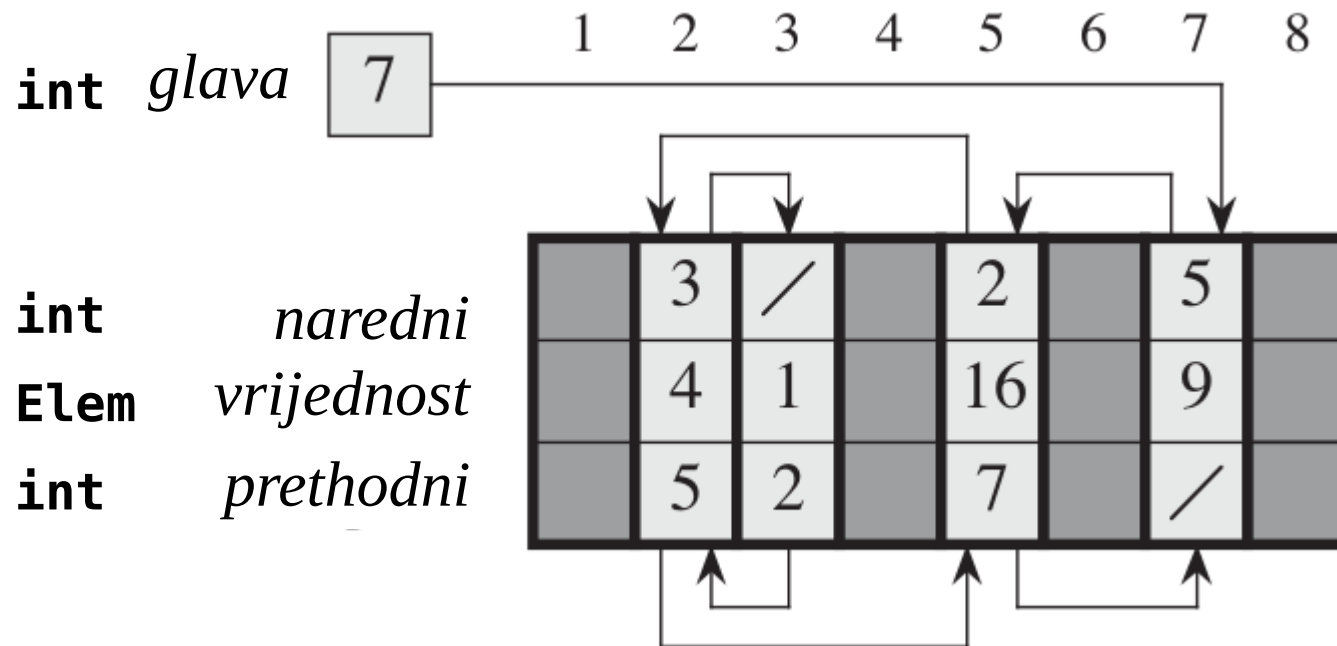
Ilustracija dodavanja novog elementa



Dvostruko povezana lista pomoću nizova (niza)

Na narednim slajdovima je dat primjer implementacije liste pomoću niza, kod koje su sve osnovne operacije složenosti $O(1)$. Lista je implementirana po uzoru na dvostruko povezanu listu.

Lista: $\langle 9, 16, 4, 1 \rangle$

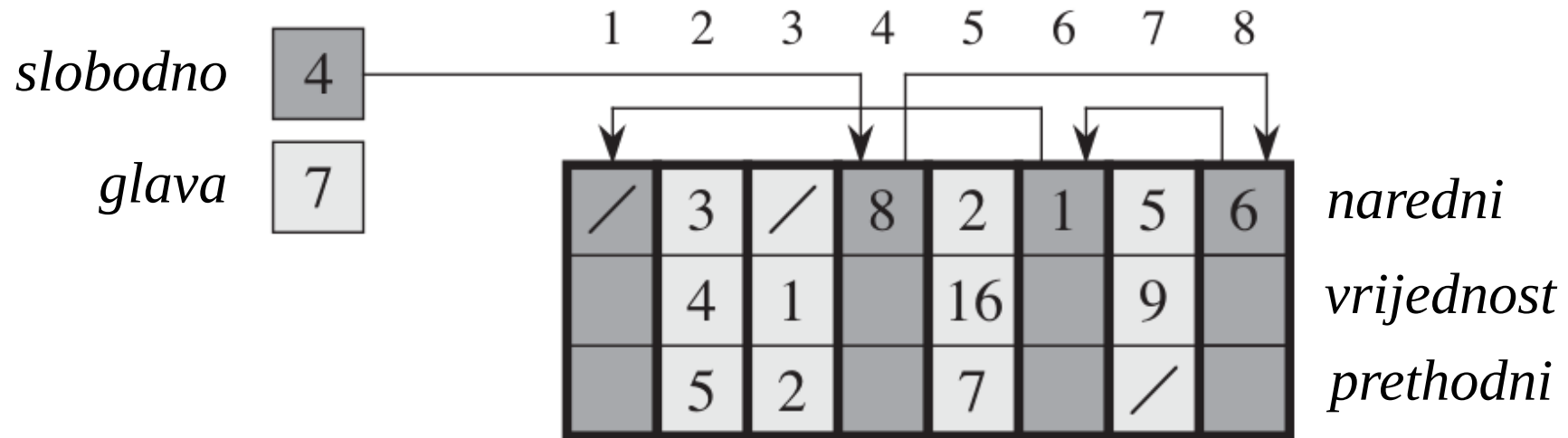


glava, naredni i prethodni čuvaju indekse.

Kako uklanjamo element iz liste?

Kako dodajemo novi element u listu?

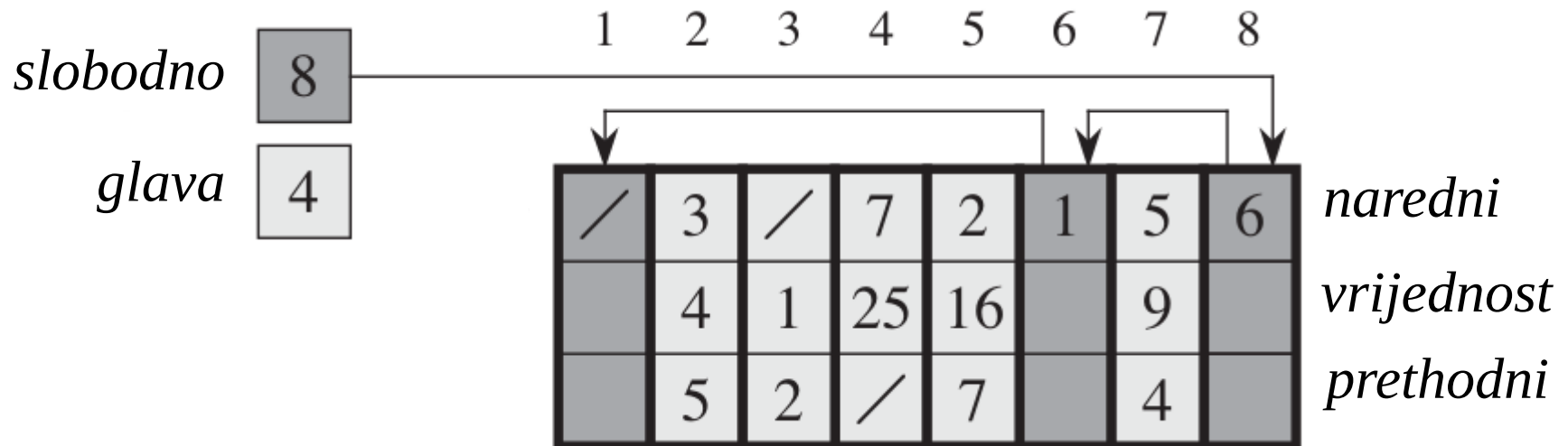
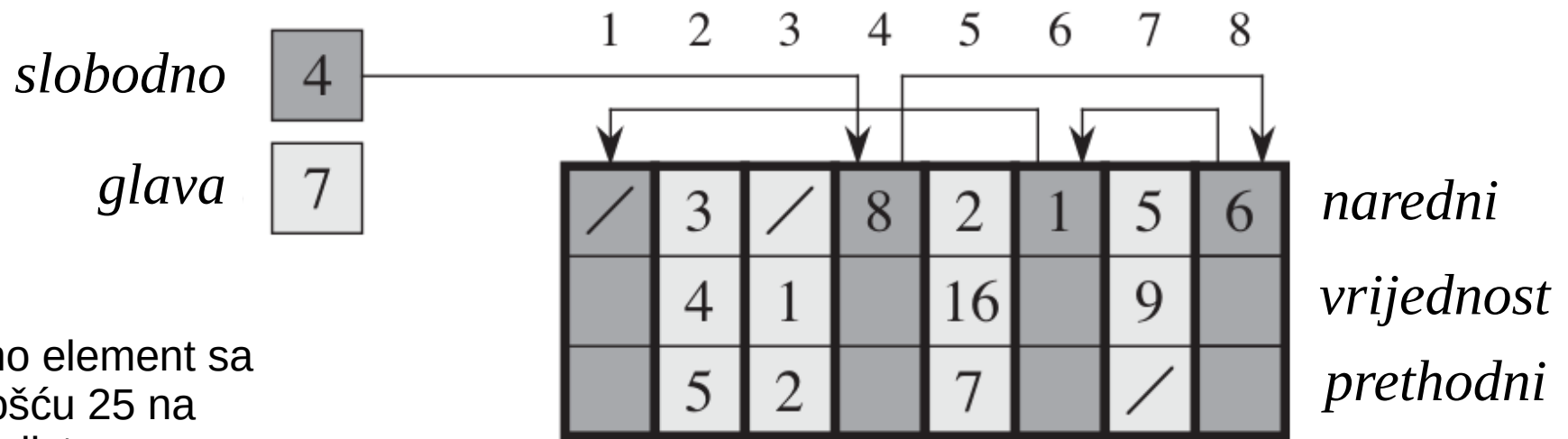
Dvostruko povezana lista pomoću nizova (niza)



slobodno predstavlja početak liste sa slobodnim elementima niza.

Ovo nam omogućava da dodavanje u listu bude $O(1)$.

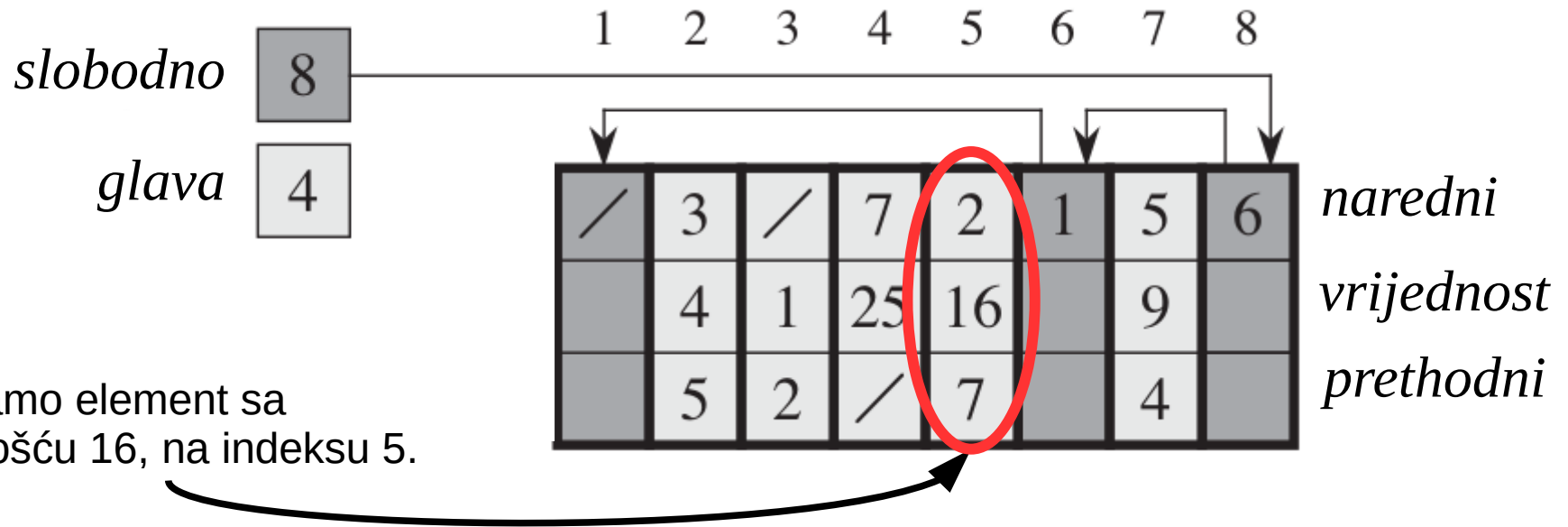
Dodavanje elementa



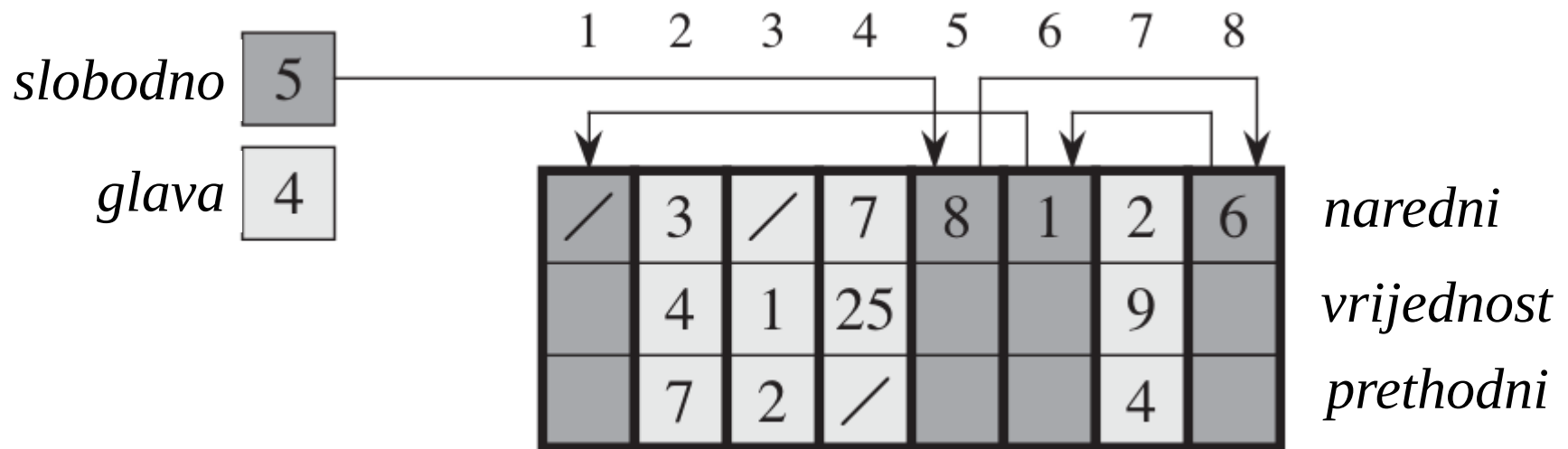
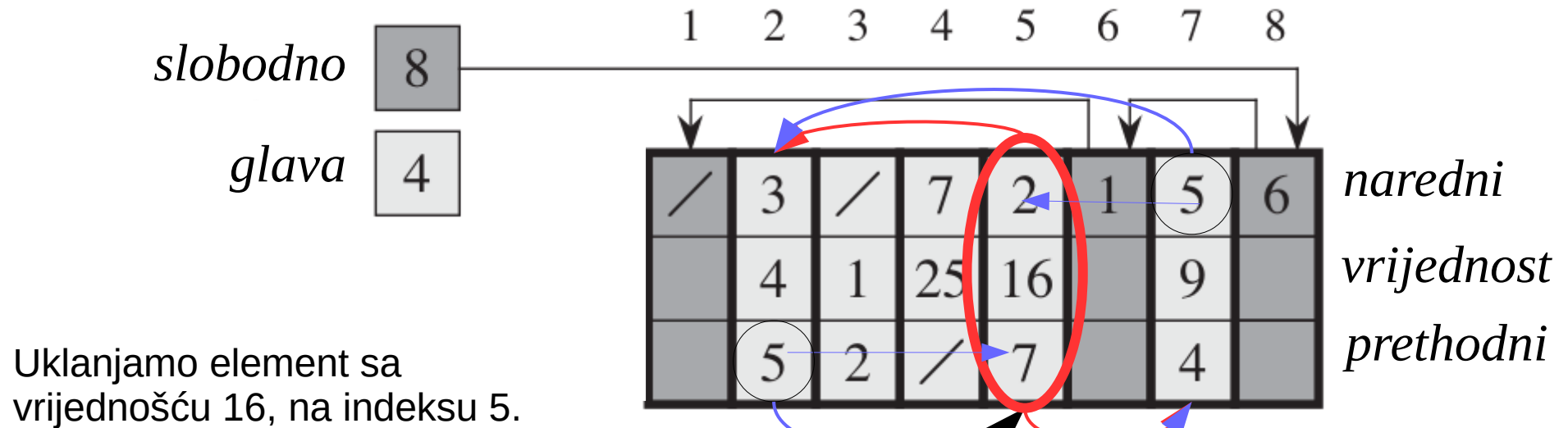
Dodavanje elementa (vrijednost 25) u listu se vrši na mjesto prvog slobodnog elementa niza (indeks slobodno), a slobodno poprima vrijednost narednog indeksa liste slobodnih elemenata.

$O(1)$

Uklanjanje elementa



Uklanjanje elementa



Uklanjanje elementa (indeks niza 5) iz liste se vrši tako što se “povežu” prethodni i naredni element elementa koji se uklanja.

Element koji se uklanja postaje prvi element liste slobodnih elemenata niza. $O(1)$

Usporedba složenosti nekih operacija za različite implementacije liste

Operacija	Sukc. memorija (niz)	Jednostruko povezana lista	Dvostruko povezana lista
Konstruktor	$O(1)$ ili $O(n)$	$O(1)$	$O(1)$
Copy konstruktor	$O(n)$	$O(n)$	$O(n)$
Destruktor	$O(1)$ ili $O(n)$	$O(n)$	$O(n)$
operator=	$O(n)$	$O(n)$	$O(n)$
Brisanje cijele liste	$O(1)$ ili $O(n)$	$O(n)$	$O(n)$
Dodavanje ispred	$O(n)$	$O(1)$	$O(1)$
Dodavanje iza	$O(n)$	$O(1)$	$O(1)$
Uklanjanje	$O(n)$	$O(1)$, zadnji $O(n)$	$O(1)$
Dodavanje na početak	$O(n)$	$O(1)$	$O(1)$
Uklanjanje sa početka	$O(n)$	$O(1)$	$O(1)$
Dodavanje na kraj	$O(1)$	$O(1)$	$O(1)$
Uklanjanje s kraja	$O(1)$	$O(n)$	$O(1)$
Traženje vrijednosti	$O(n)$	$O(n)$	$O(n)$
Marker prema naprijed	$O(1)$	$O(1)$	$O(1)$
Marker unazad	$O(1)$	$O(n)$	$O(1)$
Pristup preko indeksa	$O(1)$	$O(n)$	$O(n)$

Liste u standardnoj biblioteci C++

- Dvostruko povezana lista:
 - Tip: `std::list<ElemType>`
 - Header: `#include <list>`
- Jednostruko povezana lista (od C++11):
 - Tip: `std::forward_list<ElemType>`
 - Header: `#include <forward_list>`
- Vektor (direktan pristup elementima []):
 - Tip: `std::vector<ElemType>`
 - Header: `#include <vector>`