

# Sekvencijalni spremnici

## Objektno programiranje - 2. vježbe

Sebastijan Horvat

Prirodoslovno-matematički fakultet,  
Sveučilište u Zagrebu

15. ožujka 2023. godine



## Sekvencijalni spremnici

- ▶ **spremnik** - sadrži kolekciju objekata određenog tipa
- ▶ sekvencijalni i asocijativni spremnici
- ▶ **sekvencijalni** - redoslijed elemenata odgovara pozicijama na koje su stavljani u spremnik
- ▶ **asocijativni** - spremaju elemente na temelju vrijednosti ključa

Osobine sekvencijalnih spremnika:

- ▶ brzi sekvencijalni pristup elementima
- ▶ različite performanse glede dodavanja/uklanjanja elementa i nesekvencijalnog (*random*) pristupa elementima



## Pregled tipova sekvencijalnih spremnika

	<i>random pristup</i>	ubacivanje/izbacivanje
<b>vector</b>	brzo	sporo (osim s kraja)
<b>deque</b>	brzo	brzo na početku i kraju
<b>list</b>	samo dvosmjerni sekvencijalni pristup	brzo
<b>forward_list</b>	samo jednosmjerni sekvencijalni pristup	brzo
<b>array</b>	brzo	fiksne veličine!
<b>string</b>	brzo	sporo (osim s kraja)

Svaki definiran u svom zaglavlju s istim imenom:

- ▶ za deque trebamo `#include <deque>` itd.

Predlošci  $\Rightarrow$  kao kod vektora treba navesti tip elemenata:

- ▶ `list<double> a;`
- ▶ `deque<list<int>> c;`
- ▶ `deque<string> b;`
- ▶ itd.



## Operacije sa spremnicima

Hijerarhija operacija sa sekvencijalnim spremnicima:

- ▶ operacije koje možemo koristiti za sve spremnike
- ▶ operacije specifične sekvencijalnim spremnicima
- ▶ operacije specifične pojedinim spremnicima

**Primjer.** Vidjeli operacije s vektorima - rade li za ostale spremnike?

- ▶ `empty`, `max_size` (koliko elemenata spremnik može sadržavati)
  - ▶ podržavaju svi spremnici
- ▶ `forward_list` ne podržava `size`
- ▶ usporedba dva spremnika (**istog tipa!**): `==` i `!=` za sve, ali `<`, `<=`, `>`, `>=` ne za neuređene asocijativne spremnike



## Iteratori

- ▶ „standardne” operacije s iteratorima podržavaju svi spremnici:
  - ▶ `*it, it->a, ++it, --it, it1 == it2, it1 != it2`
- ▶ jedina iznimka: **forward\_list ne podržava --it**
- ▶ aritmetika iteratora samo za `string, vector, deque, array`

**Primjer.** Funkcija za ispis elemenata za objekt tipa `deque<int>`:

```
void ispis(deque<int> g) {
    deque<int>::iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';
}
```

Uočite kojeg je tipa `it`!

## Obrnuti iteratori (*reverse iterators*)

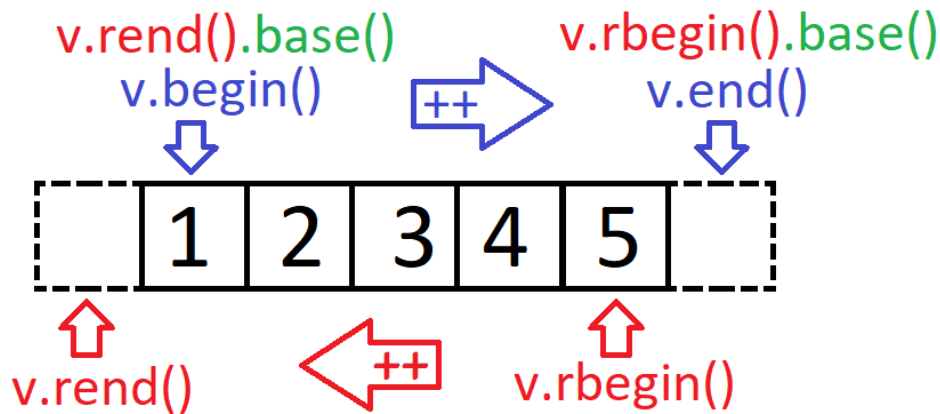
- ▶ prolazak spremnikom u **obratnom smjeru**
- ▶ ne možemo koristiti za `forward_list`
- ▶ okreću i značenje operacija s iteratorima - npr. `++` na obrnutim iteratorima daje prethodni element
- ▶ koriste **`rbegin()`** i **`rend()`** za kraj i početak spremnika
- ▶ za **`const_reverse_iterator`** (samo čitanje!): **`crbegin()`** i **`crend()`**

**Primjer.**

```
vector<int> v{1, 2, 3, 4, 5};
vector<int>::reverse_iterator it;
for (it = v.rbegin(); it != v.rend(); ++it)
    cout << *it;
```

## Obrnuti u „obični” iterator

- ▶ obrnuti iterator pretvaramo u „obični” preko `.base()`
- ▶ obrnuti referencira jedan element iza `base()` iteratora



**Primjer.** Odredite što se ispiše:

```
vector<int> v{1, 2, 3, 4, 5};  
vector<int>::reverse_iterator it;  
for(it = v.rbegin(); *it != 2; ++it);  
cout << *(it.base()) << endl;
```



## Inicijalizacija sekvencijalnih spremnika

- ▶ kao kod vektora - defaultni konstruktor stvara prazni spremnik (ne vrijedi za `array`)
- ▶ pomoću iteratora (početak i jedan iza zadnjeg) možemo kopirati dio spremnika (ponovo, ne vrijedi za `array`)
  - ▶ u tom slučaju elementi se ne moraju podudarati - elementi koje kopiramo moraju se moći pretvoriti u elemente našeg spremnika

**Primjer.**

```
list<string> imena = {"John", "Alice", "Mary"};  
vector<const char*> v = {"a", "ab", "cd"};  
list<string> list2(imena); ✓  
deque<string> d(imena); ✗  
vector<string> rijeci(v); ✗  
forward_list<string> r(v.begin(), v.end()); ✓
```



## std::array ima fiksnu veličinu

- ▶ osim tipa elemenata, treba specificirati i veličinu

### Primjer.

```
array<int, 10> a = {42};           ✓  
array<string, 20> b;              ✓  
array<int, 10>::size_type i;      ✓  
array<int>::size_type j;         ✗
```

- ▶ b ima svih 20 elemenata defaultno inicijaliziranih,
- ▶ a ima prvi element 42 (ostalih devet je 0)
- ▶ pri kopiranju se **tipovi moraju podudarati**

### Primjer. Je li sljedeći kod ispravan?

```
array<int, 10> a1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
array<int, 10> a2 = {0};  
array<int, 11> a3 = {0, 1, 2};  
a1 = a2;  
a1 = a3;
```



## Promjena veličine spremnika - **resize**

- ▶ nije moguće za `array` (fiksne veličine!)
- ▶ slučajevi ovisno o traženoj i trenutnoj veličini spremnika:
  - ▶ trenutna > tražena - brišu se elementi s kraja spremnika
  - ▶ trenutna < tražena - dodaju se elementi na kraj spremnika
- ▶ opcionalan argument za inicijalizaciju dodanih elemenata

### Primjer.

```
list<int> li(10, 42);  
li.resize(15);  
li.resize(25, -1);  
li.resize(5);
```



## assign

- ▶ samo za sekvencijalne spremnike (bez array)
- ▶ za razliku od kopiranja, ne moraju biti istog tipa (konverzija!)

### Primjer.

```
list<string> a;  
vector<const char*> v;  
a = v; ✗  
a.assign(v.cbegin(), v.cend()); ✓
```

- ▶ raspon pomoću iteratora ili broj i vrijednost elementa

### Primjer. Odredite što se ispiše!

```
deque<int> a = {1,2,3};  
vector<double> b = {3.2,5.7};  
b.assign(2,3.4);  
a.assign(b.begin(),b.end());  
for(auto it = a.begin(); it != a.end(); ++it)  
    cout << *it << endl;
```



## swap operacija

- ▶ zamjenjuje sadržaj dva spremnika **istog tipa**

### Primjer.

```
vector<string> v1(10);  
vector<string> v2(24);  
swap(v1, v2);
```

⇒ v1 sad sadrži 24 stringa, a v2 10 stringova

- ▶ zamjena dva array-a: zamjenjuje elemente ( $\mathcal{O}(n)$ )
- ▶ za ostale spremnike: ne zamjenjuje elemente (nema kopiranja, brisanja, umetanja), nego zamjenjuje interne strukture ( $\mathcal{O}(1)$ )



## deque (*Double Ended Queue*)

- ▶ a.t.p. `Queue` iz kolegija SPA - ubacivanje na kraj, izbacivanje s početka (kao red u stvarnom životu)
- ▶ ovdje ubacivanje/izbacivanje s oba kraja (u  $\mathcal{O}(1)$ )
- ▶ za razliku od vektora:
  - ▶ ne moraju zauzimati neprekinuti komad memorije
  - ▶ dodatno ima operacije ubacivanja, izbacivanja, te dohvaćanja elementa s početka i s kraja (za dohvaćanje elementa na određenom indeksu koristimo `at()`)
- ▶ istaknimo da se kod ubacivanja ubacuje kopija elementa

### Primjer.

```
deque<int> a;  
a.pop_front();  
a.pop_back();  
a.push_back(10);  
a.push_front(20);  
cout << a.at(1);
```



## Operacije pristupa vraćaju reference

### Primjer.

```
deque<string> v = {"a", "bc"};  
v.front() = "d"; //mijenja v[0]  
v[1] = "f"; //mijenja v[1]  
auto &p = v.back();  
p = "cd"; //mijenja v[1]  
auto s = v.back();  
s = "ab"; //ne mijenja v[1]
```

**Napomena.** indeks vs. `at` - u gornjem primjeru bi `v[2]` izazvao *run-time error*, dok bi `v.at(2)` izbacio `out_of_range` iznimku



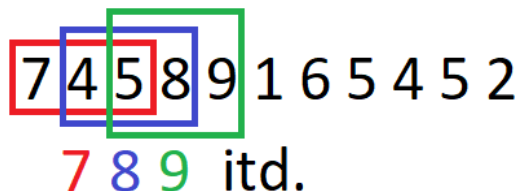
## Zadatak

Učitajte od korisnika  $k \in \mathbb{N}$ . Zatim učitavajte brojeve (do EOF) i za svakih  $k$  uzastopnih brojeva unosa ispišite najveći među njima. Riješite zadatak korištenjem deque.

### Primjeri.

Unos:	3
	7 4 5 8 9 1 6 5 4 5 2
Ispis:	7 8 9 9 9 6 6 5 5
Unos:	4
	1 2 3
Ispis:	

Ilustracija za prvi primjer:



(U drugom primjeru ništa se ne ispiše jer nije dovoljno brojeva uneseno.)

**Napomena.** Funkcija `std::max_element` definirana u zaglavlju `algorithm` za prosljeđene iteratore  $a$  i  $b$  vraća iterator koji pokazuje na element s najvećom vrijednosti u rasponu  $[a,b)$  (treći (opcionalni) argument je funkcija za uspoređivanje dva elementa).



## Preusmjeravanje u datoteku

- ▶ testiranje programa može biti naporno ako puno puta unosimo isti (veći) broj podataka
- ▶ većina OS-ova dopušta *file redirection* - omogućava da ulaz stavimo u jednu datoteku, a izlaz da se ispiše u drugu
- ▶ primjer upotrebe: ako s  $\$$  označimo *system prompt*, te imamo kompajlirani program u izvršivu datoteku `zadatak` (ili `zadatak.exe`):

```
 $\$$  ./zadatak <ulaz >izlaz
```

- ▶ ulaz čita iz datoteke `ulaz`, a rezultat ispiše u datoteku `izlaz`

```
sehorva@DESK:~/OPCPP$ g++ prvi.cpp -std=c++11 -o prog
sehorva@DESK:~/OPCPP$ cat > ulaz
3
7 4 5 8 9 1 6 5 4 5 2
sehorva@DESK:~/OPCPP$ ./prog <ulaz
8 9 9 9 6 6
sehorva@DESK:~/OPCPP$
```







## Primjer.

- ▶ Što radi sljedeći kod?
- ▶ Je li vektor bio dobar odabir ili bi bilo bolje izabrati nešto drugo za  $v$ ? Zašto?

```
string r;  
vector<string> v;  
auto it = v.begin();  
while(cin >> r)  
    it = v.insert(it, r);  
for(auto r : v)  
    cout << r << endl;
```



## Emplace operacije

- ▶ `emplace_front`, `emplace`, `emplace_back` imaju uloge kao `push_front`, `insert`, `push_back`
- ▶ razlika: **konstruiraju, a ne kopiraju** elemente
- ⇒ imaju argumente koji se prosljeđuju konstruktoru

## Primjer.

```
vector<string> v;  
v.push_back(5, 'v');           X  
v.push_back(string(5, 'v'));  ✓  
v.emplace_back(5, 'v');       ✓
```



## Uklanjanje elemenata

- ▶ za `vector` i `string` nemamo `pop_front`
- ▶ za `forward_list` nemamo `pop_back`
- ▶ općenito, **sve operacije koje mijenjaju veličinu spremnika, array ne podržava**
- ▶ `clear` - uklanja sve elemente iz spremnika
- ▶ sve gore navedene funkcije vraćaju `void`

### Primjer.

```
deque<string> v = {"a", "bc"};  
v.clear();
```

- ▶ za uklanjanje elemen(a)ta unutar spremnika koristimo `erase` (ne radi za `forward_list`!)

Prije uklanjanja elementa, treba biti siguran da on postoji!

## Uklanjanje elemenata iz spremnika (`erase`)

- ▶ navodimo iterator za element koji uklanjamo ili par iteratora za raspon (prvi i „jedan iza” zadnjeg koji uklanjamo)
- ▶ oba ta oblika `erase` vraćaju iterator koji se odnosi na lokaciju **nakon** (zadnjeg) obrisano elementa

**Primjer.** Uklanjanje svih neparnih elemenata liste:

```
list<int> li = {0,1,2,3,4,5};  
auto it = li.begin();  
while (it != li.end()) {  
    if (*it % 2) {  
        it = li.erase(it);  
    } else {  
        ++it;  
    }  
}
```

## forward\_list

- ▶ **jednostruko vezana lista** ⇒ ima posebne operacije  
(Podsjetnik: Zašto nije jednostavno dod./izbac. iz jednostruko vezane liste?)
- ⇒ nemamo `insert`, `emplace`, `erase`, nego imamo  
**`insert_after`**, **`emplace_after`**, **`erase_after`**
- ▶ kako bi primjerice uklonili 3. po redu element, moramo pozvati `erase_after` na iteratoru za **prethodni** 2. po redu element
- ⇒ **`before_begin`** vraća iterator „prije prvog” (kako bi mogli ubacivati/uklanjati s početka; za odgovarajući `const_iterator` imamo **`cbefore_begin`**)

**Primjer.** Odredite sadržaj jednostruko vezane liste:

```
forward_list<int> li = {1, 2, 3};  
auto it = li.before_begin();  
li.insert_after(it, 2, 5);
```



## Primjer. (Brisanje neparnih elemenata)

```
forward_list<int> fli = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
auto preth = fli.before_begin();  
auto tren = fli.begin();  
while (tren != fli.end()) {  
    if (*tren % 2)  
        tren = fli.erase_after(preth);  
    else {  
        preth = tren;  
        ++tren;  
    }  
}
```

- ▶ koristimo dva iteratora:
  - ▶ `tren` - za element koji provjeravamo
  - ▶ `preth` - za prethodnik tog elementa



## Oprez: Iteratori i mijenjanje spremnika!

Iteratori, pokazivači i reference (naravno ne za uklonjene elemente!) pri mijenjanju spremnika:

- ▶ za `list` i `forward_list` ostaju valjani (prethodni primjer!)
- ▶ za `vector` i `string`
  - ▶ pri dodavanju elementa moguća realokacije
  - ▶ pri uklanjanju, valjano sve do točke uklanjanja
- ▶ za `deque`:
  - ▶ nisu valjani ako dodajemo/uklanjamo element(e) unutar spremnika (tj. ne s početka ni s kraja)
  - ▶ dodavanje na početak/kraj - samo iteratori nisu valjani
  - ▶ ako brišemo s početka, sve ok, a ako brišemo s kraja, tada samo „jedan iza zadnjeg” iterator nije valjan

**Primjer.** Je li sljedeće ispravno (ako želimo ukloniti sve  $> 1$ )?

```
deque<int> d = {1,2,3};
for(auto it = d.begin(); it != d.end(); ++it)
    if(*it > 1)
        d.erase(it);
```



## Zadatak

Učitajte od korisnika prirodne brojeve (do EOF), te ih spremite u `deque`. Napišite kod koji iz njega uklanja sve parne brojeve, a udvostručava sve neparne. Ispišite dobiveni `deque`.

**Primjer.**

Ulaz: 3 2 4 5 3 6 4 2

Izlaz: 3 3 5 5 3 3

**Uputa.** Što vraćaju `insert`, odnosno `erase`?



## Malo o tome kako `vector` „raste”

- ▶ zbog brzog pristupa elementima, zauzima neprekinut komad memorije (isto za `string`)
- ▶ ako novi element ne stane, realocira se cijeli taj komad
- ▶ kako bi ta spora operacija bila što rjeđa, obično alokira malo više od potrebnog
- ▶ informacije i upravljanje veličinom spremnika:
  - ▶ broj elemenata: `size`,
  - ▶ maksimalan broj elemenata prije realokacije: `capacity`,
  - ▶ micanje „viška” memorije: `shrink_to_fit`,
  - ▶ rezerviranje memorije za najmanje `n` elemenata: `reserve(n)`.

## Primjer

```
vector<int> v = {1,2,3};
cout << v.size() << endl //3
      << v.capacity() << endl; //3
v.push_back(4);
cout << v.capacity() << endl; //6
v.shrink_to_fit();
cout << v.capacity() << endl; //4
v.reserve(10);
cout << v.capacity() << endl; //10
```

## Adapteri za sekvencijalne spremnike

- ▶ **adapter** - mehanizam koji omogućava da se jedna stvar ponaša poput druge
- ▶ za sekvencijalne spremnike: **stack**, **queue**, **priority\_queue**
- ▶ primjerice, `stack` adaptor uzima sekvencijalni spremnik i omogućuje da s njime radimo kao sa stogom (`stack`)
- ▶ **dva konstruktora**:
  - ▶ defaultni (stvara prazni objekt): `stack<int> s;`
  - ▶ kopiranjem spremnika: `stack<int> d(deq);`  
(pri čemu je `deq` tipa `deq<int>`)
- ▶ operacije koje svi adaptori spremnika imaju zajedničke:
  - ▶ `empty`, `size`
  - ▶ `swap` (za njega svi tipovi - adapter i impl. - moraju biti isti!)
  - ▶ relacijski operatori (`==`, `!=`, `<`, `<=`, `>`, `>=`) - usp. spremnika ispod
- ▶ defaultna implementacije može se promijeniti korištenjem drugog argumenta pri stvaranju adaptoru

**Primjer.** `stack<string, vector<string>> s1;`  
`stack<string, vector<string>> s2 (svektor);`



29/32

## Adaptor za stog

**Primjer.**

```
stack<string> s;
string rijec;
while(cin >> rijec)
    s.push(rijec);
cout << s.size() << endl;
s.emplace(10, '-');
while(!s.empty()){
    cout << s.top() << endl;
    s.pop();
}
```

- ▶ potrebno: **#include<stack>**
- ▶ po defaultu impl. na `deque`, a još može i `list` ili `vector`
- ▶ iako, primjerice, `push` poziva `push_back` na `deque` objektu koji je ispod, ne možemo sami koristiti `push_back` operaciju za stog!



30/32

## Adaptor za red (**queue**)

- ▶ **queue** zaglavlje
- ▶ FIFO princip spremanja i dohvaćanja elemenata
- ▶ može impl. na deque (default) ili vector

```
queue<string> red;
string rijec;
while(cin >> rijec)
    red.push(rijec);
cout << red.size() << endl;
red.emplace(10, '-');
while(!red.empty()){
    cout << red.front() << endl;
    red.pop();
}
```



## Adaptor za prioritetni red (**priority\_queue**)

- ▶ također zaglavlje **queue**; default. impl. vector, može i deque
- ▶ prioritet među elementima (prije oni s većim prioritetom)
- ▶ po defaultu se za prioritet koristi < operator (može i [drugačije](#))

```
priority_queue<string> pr;
string rijec;
while(cin >> rijec)
    pr.push(rijec);
cout << pr.size() << endl;
pr.emplace(10, '-');
while(!pr.empty()){
    cout << pr.top() << endl;
    pr.pop();
}
```

