Sekvencijalni spremnici

Objektno programiranje (C++) - 2. vježbe

dr. sc. Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

13. ožujka 2024. godine

Sekvencijalni spremnici

- **spremnik** - sadrži kolekciju objekata određenog tipa
- spremnike dijelimo na sekvencijalne i asocijativne
  - **sekvencijalni spremnici** - redoslijed elemenata odgovara pozicijama na koje su stavljeni u spremnik
  - **asocijativni spremnici** - 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

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

Svaki spremnik je definiran u svom zaglavlju s istim imenom:

- primjerice, za `deque` trebamo `#include <deque>`

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

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

Operacije sa spremnicima - hijerarhija operacija

Hijerarhija operacija sa sekvencijalnim spremnicima:

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

Primjer 1. Vidjeli smo operacije koje podržavaju vektori - podržavaju li ih i ostali spremnici?

- `empty`, `max_size` (koliko elemenata spremnik može sadržavati)
  - podržavaju ih svi spremnici
- `forward_list` **ne podržava** `size`
- **usporedba dva spremnika** (**istog tipa!**): `==` i `!=` možemo koristiti za sve spremnike, no `<`, `<=`, `>`, `>=` ne možemo koristiti 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 2. 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*)

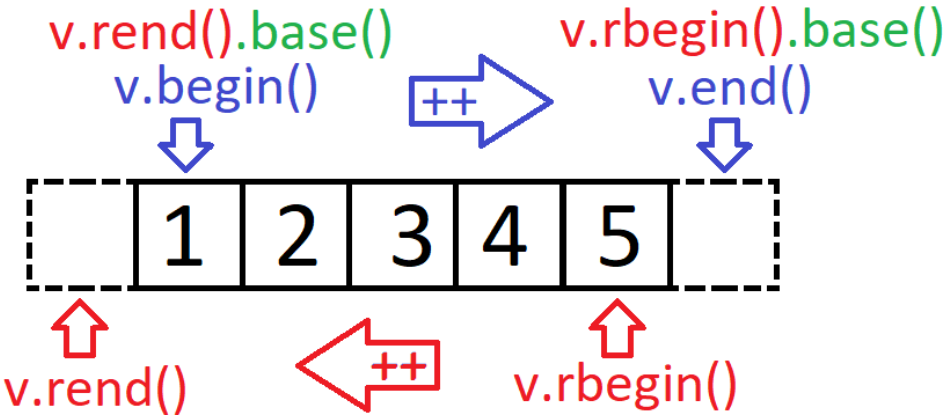
- prolazak spremnikom u **obratnom smjeru**
- ne možemo koristiti za `forward_list`
- okreću i značenje operacija s iteratorima - primjerice, `++` na obrnutim iteratorima daje prethodni element
- koriste **`rbegin()`** i **`rend()`** za kraj i početak spremnika
- za **`const_reverse_iterator`** (samo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

Primjer 3.

```
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 - samostalno proučiti

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



Primjer 4. Sljedeći dio koda ispisuje broj 3:

```
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 5.

```
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 6.

```
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 7. 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` - samostalno proučiti

- nije moguće za `array` (on je 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 8.

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

`assign` - samostalno proučiti

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

Primjer 9.

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

- navodimo raspon pomoću iteratora ili broj i vrijednost elemenata

Primjer 10. 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 - samostalno proučiti

- zamjenjuje sadržaj dva spremnika **istog tipa**

Primjer 11.

```
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 12.

```
deque<int> a = {1, 2};
a.pop_front();
a.pop_back();
a.push_back(10);
a.push_front(20);
cout << a.at(1);
```


Operacije pristupa vraćaju reference

Primjer 13.

```
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

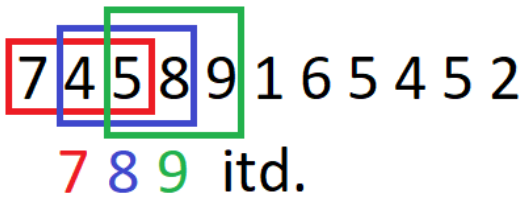
Zadatak 1.

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` spremnika.

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 proslijeđene 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$
```

Dodavanje elementa na određeno mjesto u spremniku

- `.insert()` za ubacivanje **ispred** danog iteratora u spremnik
- za `vector`, `deque`, `list` i `string`

Primjer 14. Sljedeće je ekvivalentno ubacivanju na početak:

```
list<string> rijeci;
rijeci.insert(rijeci.begin(), "jedan");
```

- iako nemamo `push_front` za vektor, ovako možemo ubacivati na početak vektora (oprez: sporo!)

Zadatak 2. Učitavajte od korisnika riječi (sve do `EOF`). Zatim ispišite sve učitane riječi zajedno s brojem njihova pojavljivanja. Riječi trebaju biti ispisane u sortiranom poretku (leksikografski). Pritom nemojte prvo učitati sve riječi pa ih sortirati, nego ih ubacujte u listu u sortiranom poretku.

Ostale mogućnosti korištenja `insert`

Možemo navesti:

- element i koliko njegovih kopija ubacujemo,
- dva iteratora za raspon ili inicijalizacijsku listu.

- vraća iterator na prvi ubačeni element (ili prvi parametar za prazni raspon)

Primjer 15. Odredite sadržaj od `d` nakon svake naredbe:

```
deque<string> d = {"a","b","c"},
  d2 = {"e","f","g"};
d.insert(d.begin(),2,"d");
d.insert(d.begin(),d2.begin()+1,d2.end());
d.insert(d.end()-1,{"m","n"});
```

Odgovor: **d,d**,a,b,c / **f,g**,d,d,a,b,c / f,g,d,d,a,b,**m,n**,c

Dodatno pitanje: Zašto ovaj kod nije ispravan za `list<string>`?

Primjer 16.

- 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 17.

```
vector<string> v;
v.push_back(5,'v');              ✗
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 18.

```
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) obrisanog elementa

Primjer 19. 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 na početak ili uklanjati s početka; za odgovarajući `const_iterator` koristimo **`cbefore_begin`**)

Primjer 20. Odredite sadržaj sljedeće jednostruko vezane liste:

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

(Rješenje: 5,5,1,2,3)

Primjer 21. 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 = flst.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 tada nema nikakvih problema, a ako brišemo s kraja, tada samo „jedan iza zadnjeg” iterator nije valjan

Iteratori i mijenjanje spremnika - primjer

Primjer 22. 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 3.

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 alocira 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 23.

```
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 spremnik): `stack<int> s;`
  - kopiranjem spremnika: `stack<int> d(deq);`
    (pri čemu je `deq` tipa `stack<int>`)
- operacije koje svi adapteri spremnika imaju zajedničke:
  - `empty`, `size`
  - `swap` (za njega svi tipovi - adapter i impl. - moraju biti isti!)
  - relacijski operatori (==, !=, <, <=, >, >=) - usp. spremnika ispod
- *defaultna* implementacija može se promijeniti korištenjem drugog argumenta pri stvaranju adaptera

Primjer 24.

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

Adapter za stog (`stack`)

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

Primjer 25.


```
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();
}
```

Adapter za red (`queue`)

- potrebno: `#include <queue>`
- može biti implementiran na `deque` (*default*) ili `vector`

- FIFO princip spremanja i dohvaćanja elemenata

Primjer 26.


```
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();
 }
```

Adapter 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

```
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();
}
```

Prioritetni red uz vlastiti komparator

Primjer 27. Promatramo uređene parove cijelih brojeva:

```
struct par {
    int x, y;
};
```

No, sljedeći kod se neće kompajlirati (Zašto?):

```
priority_queue<par, vector<par>> pred;
pred.push({4,2});
pred.push({4,5});
pred.push({2,3});
pred.push({1,3});

while(!pred.empty()) {
    cout << "\t(" << pred.top().x << ","
         << pred.top().y << ")" << endl;
    pred.pop();
}
```

Pisanje vlastitog komparatora

- treći (opcionalni) parametar predloška prioritetnog reda je komparator:

```
priority_queue<par, vector<par>, Usporedi> pred;
```

- radi se o funkcijskom objektu (funktoru) - `operator()` vraća `true` ako su `prvi` i `drugi` u ispravnom poretku, inače `false` (što znači da je potrebna njihova zamjenu za ispravan poredak)

```
struct Usporedi {
   bool operator()(par prvi, par drugi) {
      //antileksikografski uređaj
      return (prvi.y < drugi.y) ||
          (prvi.y == drugi.y && prvi.x < drugi.x);
   }
};
```

Podsjetnik - RP1: *Objekt klase koja ima nadograđen* `operator()` *zovemo funkcijski objekt ili funktor.*