



STRUKTURE PODATAKA I ALGORITMI
Predavanje 10

Ishod 3

1

Homework 01

- <https://tinyurl.com/3875uzzy>



Strana • 2



2

Logaritamska složenost

- Tvrdnja: ako binarno stablo ima n čvorova, onda je dubina stabla $h \geq \log_2 n$
- Dokaz:
 - Kako 15 čvorova možemo smjestiti u binarno stablo tako da dubina bude što manja?
 - Tako da napravimo savršeno binarno stablo dubine 3
 - Ekvivalentno, ako imamo savršeno stablo dubine 3, u njega stane 15 čvorova
- Zanima nas sljedeće: ako imamo stablo dubine h , koliko čvorova stane u njega?

Strana * 3



3

Dokaz logaritamske složenosti (1/2)

- U stablo dubine 0 stane: $2^1 - 1$ (1 čvor)
- U stablo dubine 1 stane: $2^2 - 1$ (3 čvora)
- U stablo dubine 2 stane: $2^3 - 1$ (7 čvorova)
- U stablo dubine 3 stane: $2^4 - 1$ (15 čvorova)
- U stablo dubine 4 stane: $2^5 - 1$ (31 čvor)
- ...
- U stablo dubine h stane: $2^{h+1} - 1$

Strana * 4



4

Dokaz logaritamske složenosti (2/2)

- Dalje računamo:

$$n = 2^{h+1} - 1$$

$$n + 1 = 2^{h+1}$$

$$\log_2(n + 1) = h + 1$$

$$h = \log_2(n + 1) - 1 \approx \log_2 n$$

- Dakle, dubina savršenog binarnog stabla je $\log n$
 - Dubina svih ostalih stabala je veća
- Algoritmi koji obrađuju na svakoj razini po jedan čvor mogu postići brzinu $\Omega(\log n)$
 - Najgori slučaj je $O(n)$ ako imamo koso stablo
 - Zanimljivo: srednji slučaj je uvijek bliži najboljem, tj. $\Theta(\log n)$

Strana • 5



5

HRPA

Strana • 6



6

Uvod

- **Hrpa** ili gomila (engl. *heap*) je struktura podataka koja zadovoljava uvjete:
 - Potpuno je binarno stablo
 - Može biti bilo kakvo potpuno stablo, ali ćemo mi promatrati samo binarna stabla
 - Vrijednost u čvoru roditelju je:
 - Uvijek veća ili jednaka svim vrijednostima djece (engl. *max-heap*), ili
 - Uvijek manja ili jednaka svim vrijednostima djece (engl. *min-heap*)
 - Napomena: primijetimo da se ništa ne kaže o vrijednostima braće
- Hrpa ima veliku važnost i učestalu primjenu u računarstvu:
 - Za implementaciju prioritetskog reda
 - Za primjenu algoritma sortiranja HEAPSORT

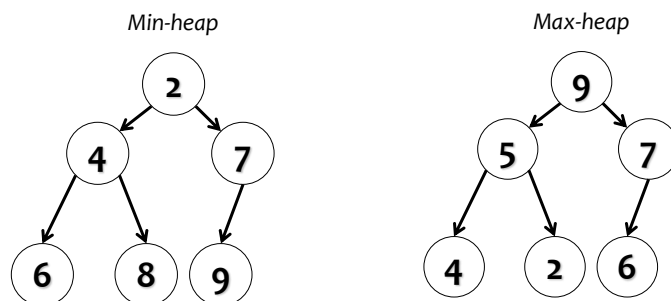
Strana • 7



7

Max-heap i min-heap

- U *min-heapu* je najmanji element stavljen u korijen stabla
- U *max-heapu* je najveći element stavljen u korijen stabla



- U ostatku predavanja ćemo promatrati *max-heap*
- *min-heap* varijanta je ekvivalentna u svemu

Strana • 8



8

IZGRADNJA POTPUNOG BINARNOG STABLA

Strana • 9



9

Izgradnja potpunog binarnog stabla (1/8)

- Želimo u potpuno binarno stablo smjestiti vrijednosti:
45, 35, 23, 27, 21, 22, 4, 19

45

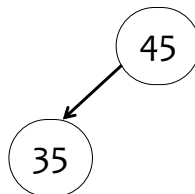
Strana • 10



10

Izgradnja potpunog binarnog stabla (2/8)

- Drugi čvor je uvijek lijevo dijete korijena



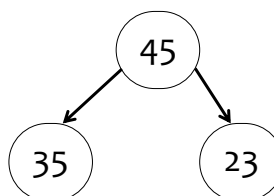
Strana • 11



11

Izgradnja potpunog binarnog stabla (3/8)

- Treći čvor je uvijek desno dijete korijena



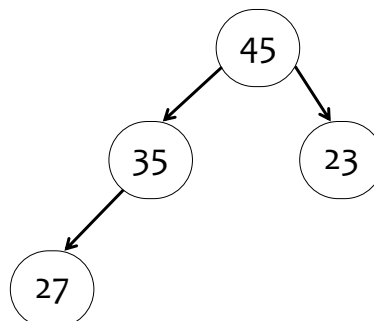
Strana • 12



12

Izgradnja potpunog binarnog stabla (4/8)

- Sljedeći čvor uvijek puni sljedeću razinu s lijeva na desno



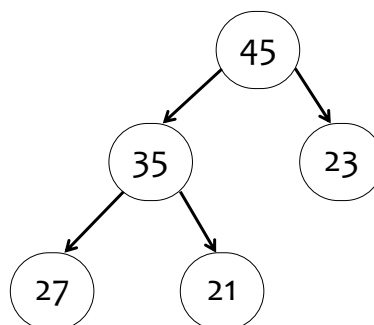
Strana • 13



13

Izgradnja potpunog binarnog stabla (5/8)

- Sljedeći čvor uvijek puni sljedeću razinu s lijeva na desno



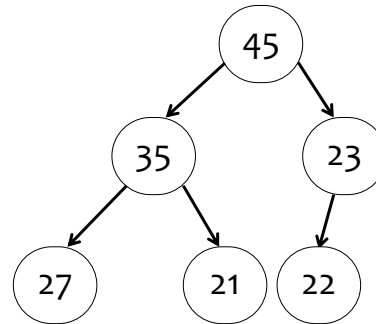
Strana • 14



14

Izgradnja potpunog binarnog stabla (6/8)

- Sljedeći čvor uvijek puni sljedeću razinu s lijeva na desno



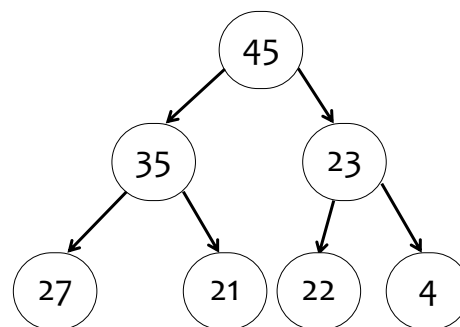
Strana • 15



15

Izgradnja potpunog binarnog stabla (7/8)

- Sljedeći čvor uvijek puni sljedeću razinu s lijeva na desno



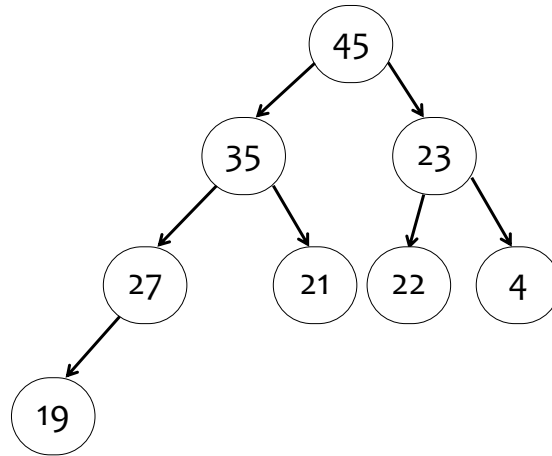
Strana • 16



16

Izgradnja potpunog binarnog stabla (8/8)

- Dobili smo potpuno binarno stablo sa 8 čvorova
- Poštujući pravilo da vrijednost roditelja mora biti \geq vrijednosti djece, dobili smo hrpu tipa *max-heap*
- Podaci su unaprijed bili tako pripremljeni
- Izgradnja ima složenost $O(n)$



17

DODAVANJE ČVORA U HRPU

Strana • 18



18

Postupak dodavanja

- Za dodavanje čvora u hrpu koristimo sljedeći postupak:
 1. Čvor inicijalno dodajemo na prvo slobodno mjesto prema prethodnim pravilima
 2. Čvor podižemo prema korijenu **zamjenom s roditeljem** sve dok ne dođe na ispravno mjesto

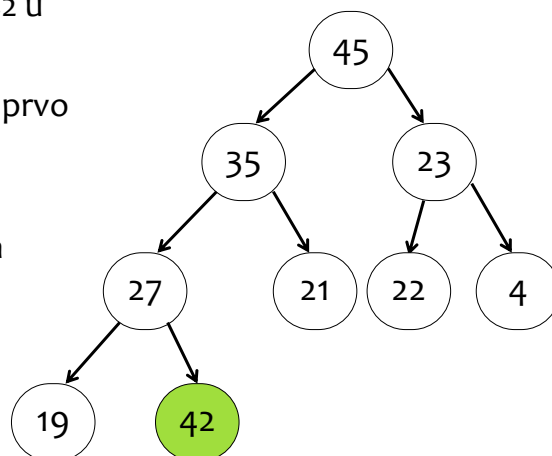
Strana • 19



19

Dodavanje čvora u hrpu (1/3)

- Dodajmo vrijednost 42 u prethodnu hrpu
- Novi čvor stavimo na prvo sljedeće mjesto
- Nadalje, održavamo svojstvo hrpe tako da novi čvor podižemo na gore, zamjenom s njegovim roditeljem, sve dok ne dođe na ispravnu lokaciju



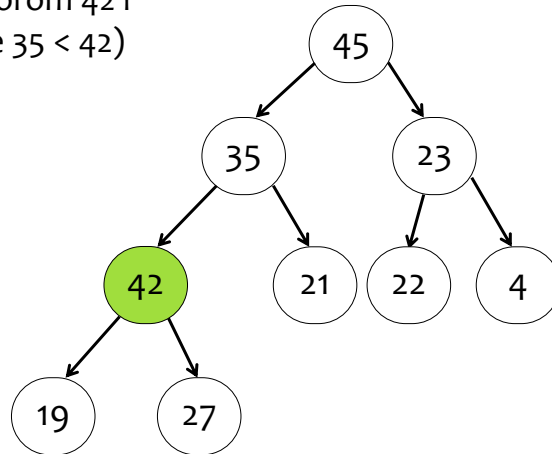
Strana • 20



20

Dodavanje čvora u hrpu (2/3)

- Nakon zamjene sa čvorom 42 i dalje nije dobro (jer je $35 < 42$)



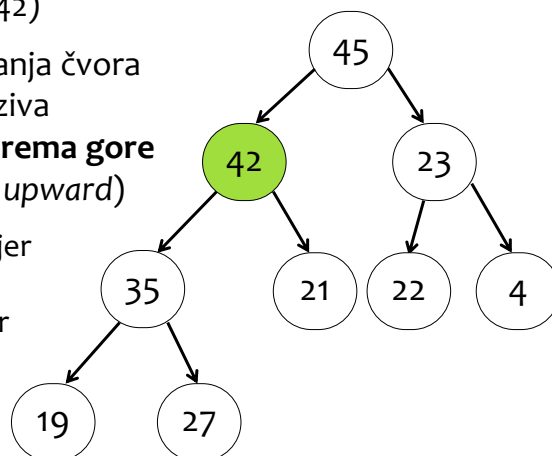
Strana • 21



21

Dodavanje čvora u hrpu (3/3)

- Sad je OK (jer je $45 > 42$)
- Ovakav proces podizanja čvora prema korijenu se naziva **preuređivanje hrpe prema gore** (engl. *reheapification upward*)
 - Složenost je $O(\log n)$ jer na svakoj razini obrađujemo po 1 čvor



Strana • 22



22

SKIDANJE S VRHA HRPE

Strana • 23



23

Postupak skidanja

- Kod hrpe uvijek čitamo/skidamo element na vrhu
 - To je najveći element na hrpi
- Uklanjanje elementa s vrha hrpe narušava samu strukturu hrpe
 - Potrebno je napraviti neku akciju kako bismo održali svojstvo hrpe
- Postupak:
 1. Uzimamo zadnji čvor i prebacujemo ga u korijen
 2. Čvor spuštamo prema listovima sve dok ne dođe na ispravno mjesto
 - Uvijek ga mijenjamo s većim djetetom (jer je *max-heap*)

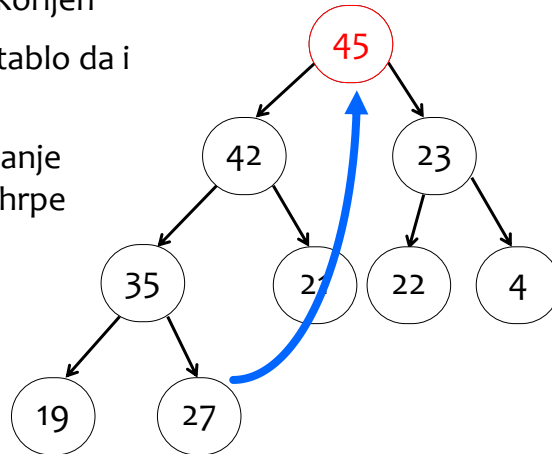
Strana • 24



24

Skidanje s vrha hrpe (1/4)

- Uzmemo i obradimo korijen
- Kako sad preurediti stablo da i dalje ostane hrpa?
- Prvi korak je prebacivanje zadnjeg čvora na vrh hrpe



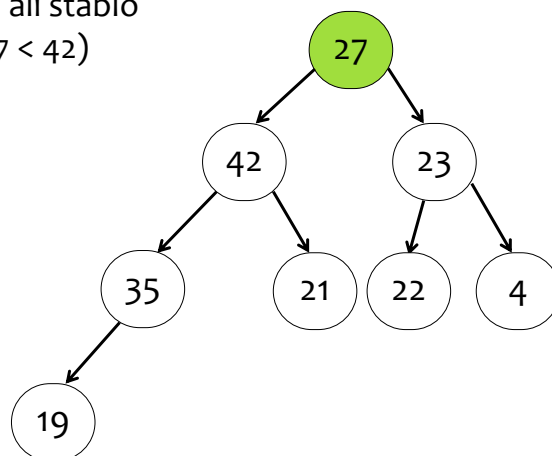
Strana • 25



25

Skidanje s vrha hrpe (2/4)

- Sad je čvor prebačen, ali stablo više nije hrpa (jer je $27 < 42$)



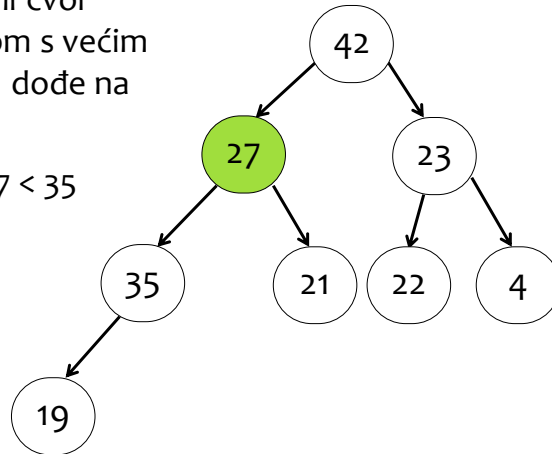
Strana • 26



26

Skidanje s vrha hrpe (3/4)

- Spuštamo premješteni čvor prema dolje, zamjenom s većim djetetom, sve dok ne dođe na ispravnu lokaciju
- Još nije dobro jer je $27 < 35$



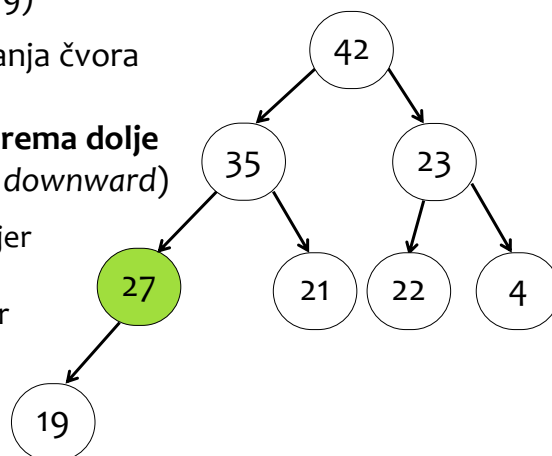
Strana • 27



27

Skidanje s vrha hrpe (4/4)

- Sad je OK (jer je $27 > 19$)
- Ovakav proces spuštanja čvora prema listu se naziva **preuređivanje hrpe prema dolje** (engl. *reheapification downward*)
 - Složenost je $O(\log n)$ jer na svakoj razini obrađujemo po 1 čvor



Strana • 28



28

Rezultat skidanja s hrpe

- Skidanjem s hrpe uništavamo hrpu
 - Jednako kao kod stoga i reda
- Rezultat su padajuće poslagani elementi
 - Za *min-heap* bi bili poslagani rastuće
- Brzina je velika jer je hrpa uvijek optimalno organizirana

Strana • 29



29

DEMO

- <http://www.cs.usfca.edu/~galles/visualization/Heap.html>
- Kreirajte *min-heap* hrpu s vrijednostima: 2, 4, 6, 8, 10, 12, 14
- Dodajte vrijednost: 5, 3, 1
- Uklonite korijen hrpe gumbom "Remove Smallest"

Strana • 30



30

PRIORITETNI RED

Strana • 31



31

Uvod

- U praksi je česta situacija da je u jednom redu potrebno definirati određene prioritete
 - Primjerice, kod liječnika se ulazi prema FIFO principu
 - Iznimka su hitni pacijenti (ili oni s vezom) jer oni imaju prednost
 - Kažemo da su to pacijenti višeg prioriteta
 - Ako postoji više pacijenata istog prioriteta, njih liječnik opet rješava po FIFO principu
 - To nije preduvjet za prioritetni red; možemo imati i prioritetne redove koji elemente istog prioriteta obrađuju nekim drugim redom

Strana • 32



32

Prioritetni red

- **Prioritetnim redom** nazivamo FIFO strukturu u kojoj svaki element ima definiran prioritet
 - Prvo se obrađuju elementi najvišeg prioriteta po FIFO principu
 - Nakon toga se na jednak način obrađuju elementi nižeg prioriteta i tako sve do najnižeg prioriteta

Strana • 33



33

Primjena

- Glavne primjene prioritetnog reda su:
 - Algoritmi traženja najkraćeg puta (računalne igre):
 - Dijkstrin algoritam
 - A* algoritam
 - Kompresiranje podataka
 - Sortiranje (*heap sort*)
 - *Task scheduler* u operacijskim sustavima

Strana • 34



34

HRPA U STL

Strana • 35



35

Uvod

- STL nudi dva načina rada s hrpom i prioritetnim redom:
 - Izravna izrada i korištenje hrpe pomoću zaglavlja `<algorithm>`
 - Nešto kompleksniji način
 - Korištenje kontejnera `priority_queue<...>` koji u stvari omata funkcionalnosti hrpe iz zaglavlja `<algorithm>`
 - Nešto jednostavniji način

Strana • 36



36

Izravna izrada i korištenje hrpe (1/3)

- Zaglavlje `<algorithm>` nudi tri funkcije za rad s hrpom:
 - `make_heap(begin, end)`
 - Preslaguje elemente u rasponu `[begin, end)` tako da formiraju *max-heap*
 - Nakon što funkcija završi, najveći element se garantirano nalazi na mjestu `begin`
 - Elementi se mogu nalaziti u polju ili objektu tipa `array<T, N>`, `vector<T>` ili `deque<T>`
 - Složenost je $O(n)$

Strana • 37



37

Primjer

```
vector<int> v = { 33, 11, 22, 88, 77, 55, 44, 33, 22, 66 };
make_heap(v.begin(), v.end());

for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl;
```

- Provjerimo je li ovo zaista hrpa

Strana • 38



38

Izravna izrada i korištenje hrpe (2/3)

- Druga funkcija za rad s hrpom:
 - `push_heap(begin, end)`
 - Smatra se da prilikom poziva funkcije hrpu čine `[begin, end - 1)` te da se novododani element nalazi na mjestu `end`
 - Funkcija uzima element na `end` i stavlja ga na odgovarajuće mjesto (preslagivanje prema gore)
 - Nakon završetka funkcije hrpu čini raspon `[begin, end)`
 - Složenost je $O(\log n)$
 - Kako možemo shvatiti ovu funkciju: „uzmi zadnji element i prebaci ga gdje treba tako da sve skupa bude hrpa”

Strana * 39



39

Primjer

```
vector<int> v = { 33, 11, 22, 88, 77, 55, 44, 33, 22, 66 };
make_heap(v.begin(), v.end());

v.push_back(99);
for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl;

push_heap(v.begin(), v.end());
for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl;
```

Strana * 40



40

Izravna izrada i korištenje hrpe (3/3)

- Treća funkcija za rad s hrpom:
 - `pop_heap(begin, end)`
 - Smatra se da prilikom poziva funkcije hrpu čine `[begin, end)`
 - Funkcija prebacuje element s pozicije `begin` na poziciju `end - 1` (preslagivanje prema dolje)
 - Nakon završetka funkcije hrpu čine `[begin, end - 1)`
 - Funkciju možemo shvatiti i ovako: „izbaci element s vrha hrpe pa presloži sve ostale tako da ovo i dalje bude hrpa, samo s jednim elementom manje”

Strana • 41



41

Primjer

```
vector<int> v = { 33, 11, 22, 88, 77, 55, 44, 33, 22, 66 };
make_heap(v.begin(), v.end());

v.push_back(99);
for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl;

push_heap(v.begin(), v.end());
for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl;

pop_heap(v.begin(), v.end());
for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
```

Strana • 42



42

PRIORITETNI RED U STL

Strana • 43



43

priority_queue<...>

- Klasa `priority_queue<...>` je kontejnerski adapter
 - Omotač oko sadržanog kontejnera koji ima svojstva hrpe
 - Sadržani kontejner može biti:
 - Vektor (podrazumijevano)
 - Deque
 - Bilo koja naša klasa koja implementira metode:
 - `empty()`
 - `size()`
 - `front()`
 - `push_back()`
 - `pop_back()`
 - Klasa mora pružati izravan pristup *i*-tom elementu (iz tog razloga lista ne može biti sadržani kontejner)

Strana • 44



44

Osnovni načini izrade prioritetnog reda

- Osnovni načini izrade prioritetnog reda su:
 - `priority_queue<int>` jedan;
 - Izrađuje prazni prioritetni red podržan vektorom
 - `priority_queue<int, deque<int>>` dva;
 - Izrađuje prazni prioritetni red podržan deque-om
 - `vector<int> v({ 11, 22, 33 });`
`priority_queue<int> tri(v.begin(), v.end());`
 - Izrađuje prioritetni red podržan vektorom i u njega kopira i slaže elemente iz raspona `[begin, end)`
- „The constructor ... calls ... `make_heap` on the range that includes all its elements ...”

Strana • 45



45

Korištenje prioritetnog reda (1/2)

- `pq.push(val)` dodaje kopiju od `val` na odgovarajuće mjesto u prioritetnom redu
 - „This member function effectively calls ... `push_back` of the underlying container object, and then reorders it to its location in the heap by calling the `push_heap` algorithm ...”
- `pq.pop()` uklanja element s najvišim prioritetom (onaj na vrhu hrpe)
 - „This member function effectively calls ... `pop_heap` algorithm to keep the heap property of `priority_queues` and then calls the member function `pop_back` of the underlying container object to remove the element”

Strana • 46



46

Korištenje prioritetnog reda (2/2)

- `pq.top()` vraća referencu na element s najvišim prioritetom (onaj na vrhu hrpe)
- `pq.size()` vraća broj elemenata u prioritetnom redu
- `pq.empty()` vraća je li prioritetni red prazan ili ne

Strana • 47



47

SLOŽENIJA PRIMJENA PRIORITETNOG REDA

Strana • 48



48

Gdje su tu prioriteti?

- Prethodni primjeri pretpostavljaju da je vrijednost koja se čuva u redu jednaka prioritetu
 - Možemo li, primjerice, u redu čuvati e-mail poruke od kojih svaka ima neki prioritet?
- Da bismo u prioritetnom redu mogli čuvati bilo kakve podatke, moramo dati odgovor na sljedeće pitanje:
 - Ako imamo dva objekta nekog tipa, koji je od njih manji?
- Najlakši način je definirati komparator:

```
struct MladjiImajuPrioritet {
    bool operator()(Osoba& o1, Osoba& o2) {
        return o1.godina_rodjenja < o2.godina_rodjenja;
    }
};
```

Strana • 49

Operator poziva funkcije



49

Složeniji način izrade prioritetnog reda

- Kad imamo definiran komparator, možemo kreirati prioritetni red na sljedeći način:

```
priority_queue<
    Osoba,
    vector<Osoba>,
    MladjiImajuPrioritet> pqosobe;
```

- Izrađuje prazni prioritetni red podržan vektorom pri čemu se za definiranje prioriteta koristi MladjiImajuPrioritet

Strana • 50



50

Zadaci

1. Neka je zadan vektor od 10 elemenata. Koristeći prioritetni red, ispišimo elemente u sortiranom redosljedu, od većih prema manjim.
2. Promijenite prethodni zadatak tako da brojevi budu ispisani od manjih prema većim (*min-heap*)
3. Napravimo program koji koristi prioritetni red za obradu zaprimljenih poruka prema prioritetima (1 = minimalni, 2 = normalni, 3 = visoki prioritet). Zapravimo nekoliko poruka pa ih obradimo ispisivanjem na ekran.

Strana • 51



51

Rješenje zadatka 1

```
vector<int> brojevi({ 17, 6, 99, 52, 11, 1, 8, 15, 7, 23 });
priority_queue<int> pq(brojevi.begin(), brojevi.end());

while (!pq.empty()) {
    cout << pq.top() << endl;
    pq.pop();
}
```

Strana • 52



52

Rješenje zadatka 2

```
#include <functional>
...
vector<int> brojevi({ 17, 6, 99, 52, 11, 1, 8, 15, 7, 23 });
priority_queue<int, vector<int>, greater<int>>
                pq(brojevi.begin(), brojevi.end());

while (!pq.empty()) {
    cout << pq.top() << endl;
    pq.pop();
}
```

Strana * 53



53

Rješenje zadatka 3 (klase)

```
struct Message {
    string subject;
    string body;
    int priority;

    Message(string subject, string body, int priority) {
        this->subject = subject;
        this->body = body;
        this->priority = priority;
    }
};

struct HigherToLowerPriorityComparator {
    bool operator() (Message& m1, Message& m2) {
        return m1.priority < m2.priority;
    }
};
```

Strana * 54



54

Rješenje zadatka 3 (main)

```

priority_queue<Message, vector<Message>,
HigherToLowerPriorityComparator> pq;
pq.push(Message("Macka lovi lopticu",
    "Pogledaj ovaj fora video :)", 1));
pq.push(Message("Sutra me nema",
    "Uzet cu jedan dan GO", 2));
pq.push(Message("Hitan sastanak",
    "Za 30 minuta, soba 204, obavezno doci.", 3));

while (!pq.empty()) {
    cout << pq.top().subject << " " << pq.top().priority <<
endl;
    pq.pop();
}

```

Strana * 55



55

Dodatni materijali

- Dodatni materijali su dostupni na:
 - Heap and priority queue as ADTs
 - <https://youtu.be/zYo-vCiUKuM>
 - Heap and priority queue in STL
 - <https://youtu.be/fOige4fHPMA>

Strana * 56



56