5. Prioritetni red, gomila i Heapsort

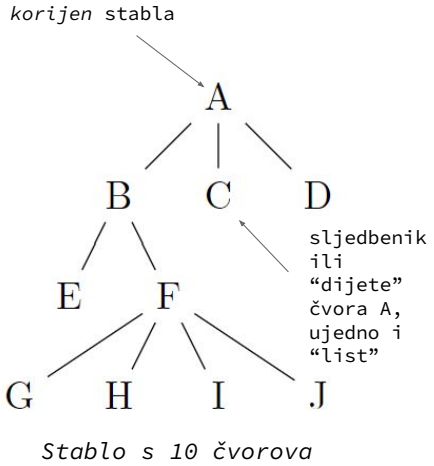
Prioritetni red

- Upotrebljava se kao prioritetni red i za Heapsort algoritam
- Prioritetni red: apstraktni tip podataka kod kojeg svaki element ima pridruženu vrijednost koja označava njegov prioritet
- Elementi se uzimaju iz reda na osnovu njihovog prioriteta, umjesto uzimanja elemenata po redu (koji je prvi ušao, prvi ide van)
- Koristi se kod sustava koji izvode operacije zavisno od njihove važnosti (prema nekom kriteriju), na primjer, izvođenje procesa kod operacijskih sustava ili poredak polijetanja ili slijetanja aviona ili obrade narudžbi

Možemo li koristit običan niz kao prioritetni red?

- Možemo, ali nije efikasno
- Primjer: (A, 4), (B, 1), (C, 8), (D, 12), (E, 5), (F, 7), …
- Kako dolazimo do elementa s najvišim prioritetom?
- Zavisi je li niz sortiran - ako nije, složenost je O(n) u najgorem slučaju, a ako je sortiran onda je $O(\log_2 n)$ (binarno pretraživanje)
- Sortirani niz bio bi efikasan ako često (ili uvijek) dodajemo ili uzimamo element od kraja
- Ako dodajemo elemente različitog prioriteta, elemente moramo pomicati da bi novi smjestili na odgovarajuće mjesto, a to je skupa operacija (O(n))
- Primjer: (A, 1), (B, 3), (C, 4), (D, 8), (E, 12), (F, 13), … da bi dodali (X, 2) moramo pomaknuti sve elemente desno od (A, 1)

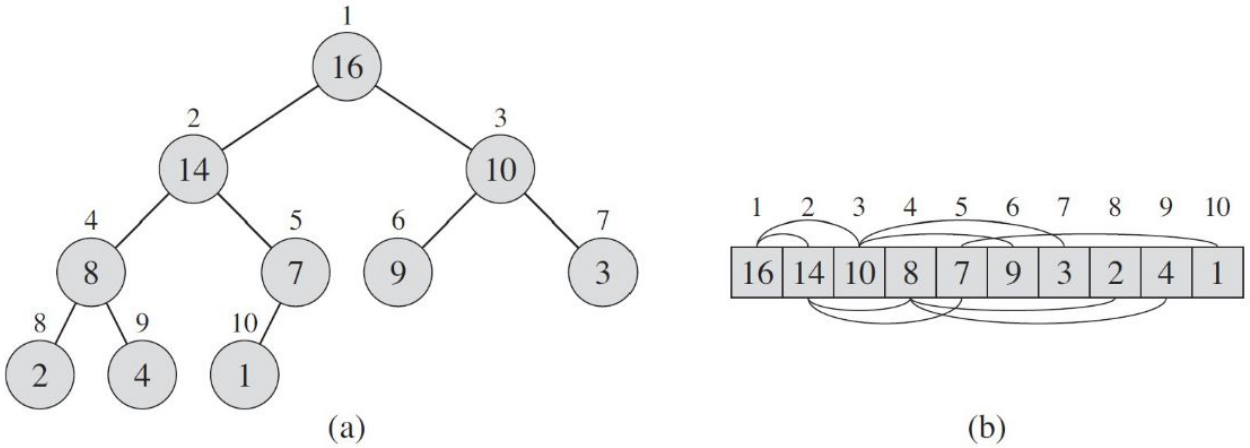
Gomila (engl. *heap*)



Stablo s 10 čvorova

- Nema veze s dinamičkom memorijom ("heap")
- Implementira se kao posebno uređen niz, prikazuje se kao stablo
- Dvije vrste gomile: *maks-gomila* i *min-gomila*
- **Maks-gomila**: svaki čvor stabla sadrži vrijednost veću ili jednaku od svojih sljedbenika (djece)
- **Min-gomila**: svaki čvor stabla sadrži vrijednost manju ili jednaku od svojih sljedbenika (djece)

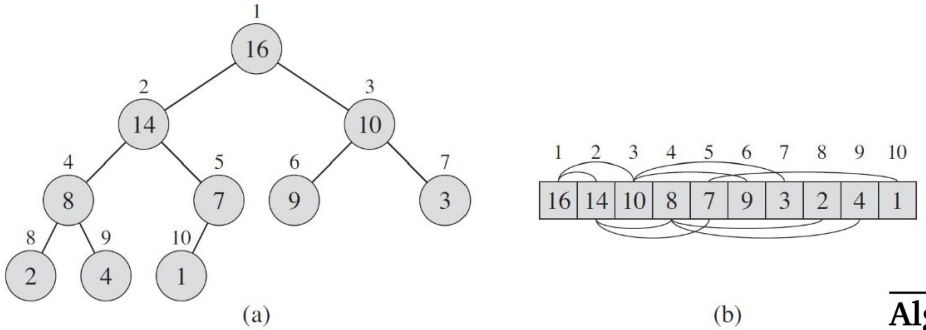
Gomila



Slika 3.1: Binarna gomila kao (a) stablo i (b) niz.

- Stablo se popunjava s lijeva na desno, s tim da donja razina može ostati nepopunjena do kraja

Gomila



Slika 3.1: Binarna gomila kao (a) stablo i (b) niz.

- Algoritmi nad gomilom rade s nizom
- **Na indeksu 1 je korijen**, lokaciju djece možemo izračunati pomoću procedura LEFT i RIGHT
- Svojstvo gomile: A[RODITELJ(i)] ≥ A[i] za maks-gomilu ili A[RODITELJ(i)] ≤ A[i] za min-gomilu
- Niz koji sadrži gomilu ima dva atributa: veličina niza i veličina gomile, za koje vrijedi
  0 ≤ niz.veličina_gomile ≤ niz.veličina_niza

Algoritam 3.8 Roditelj čvora i.

```
1: function RODITELJ(i)
2:     return ⌊i/2⌋
3: end function
```

Algoritam 3.9 Lijevo dijete čvora i.

```
1: function LIJEVO(i)
2:     return 2i
3: end function
```

Algoritam 3.10 Desno dijete čvora i.

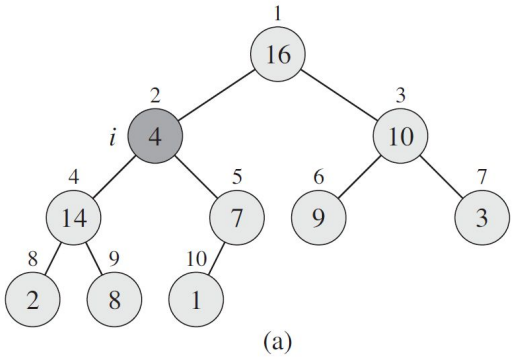
```
1: function DESNO(i)
2:     return 2i + 1
3: end function
```

Operacije nad gomilom

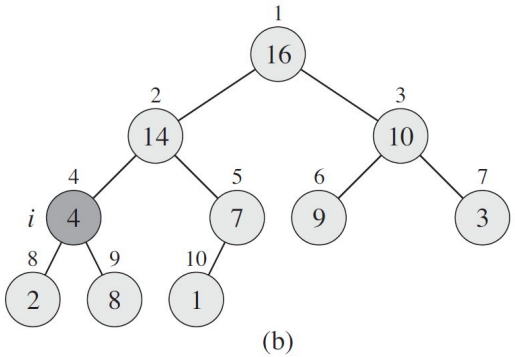
- MAKS-GOMILANJE (engl. *max-heapify*) - služi za održavanje svojstva maks-gomile (nakon brisanja ili dodavanja elementa), radi u $O(h)$, gdje je h visina polaznog čvora
- IZGRADI-MAKS-GOMILU - stvara maks-gomilu na osnovu zadanog neuređenog ulaznog niza i radi u $O(n)$ vremenu

Održavanje svojstva gomile: algoritam MAKS-GOMILANJE

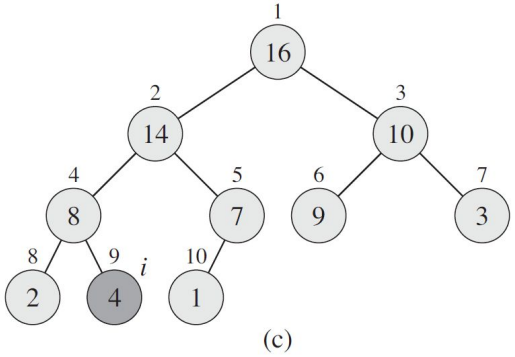
- MAKS-GOMILANJE(A, 2)
- veličina gomile = 10



(a)



(b)



(c)

Održavanje svojstva gomile: algoritam MAKS-GOMILANJE

Algoritam 3.11 Algoritam za održavanje svojstva gomile [7]. Parametar A je ulazni niz, a i indeks na kojem se nalazi čvor od kojeg se algoritam izvodi.

```
1:  procedure MAKS-GOMILANJE(A, i)
2:      l ← LIJEVO(i)
3:      d ← DESNO(i)
4:      if l ≤ A.velicina_gomile and A[l] > A[i] then
5:          najveci ← l
6:      else
7:          najveci ← i
8:      end if
9:      if d ≤ A.velicina_gomile and A[d] > A[najveci] then
10:         najveci ← d
11:     end if
12:     if najveci ≠ i then
13:         zamijeni A[i] i A[najveci]
14:         MAKS-GOMILANJE(A, najveci)
15:     end if
16: end procedure
```

ne mora postojati lijevo i desno dijete

- nađi najveći između čvora i lijevog sljedbenika, a onda između tog najvećeg čvora i desnog sljedbenika
- *najveći* sadrži indeks, ne vrijednost čvora

ako je čvor *i* već na ispravnom mjestu, prekidamo

sada polazimo od indeksa najvećeg čvora

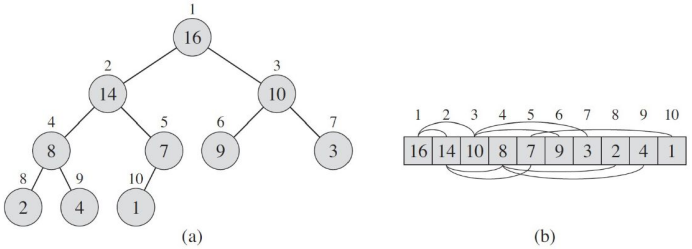
Izgradnja maks-gomile: algoritam IZGRADI-MAKS-GOMILU

Algoritam 3.12 Procedura za izgradnju gomile [7]. Parametar A je ulazni niz.

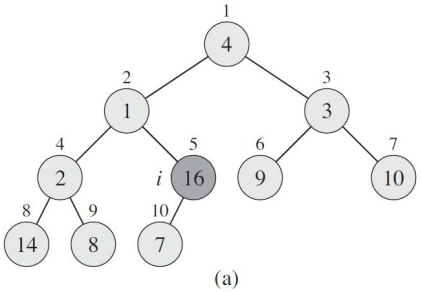
```
1: procedure IZGRADI-MAKS-GOMILU(A, n)
2:     A.velicina_gomile ← n
3:     for i ← ⌊n/2⌋ downto 1 do
4:         MAKS-GOMILANJE(A, i)
5:     end for
6: end procedure
```

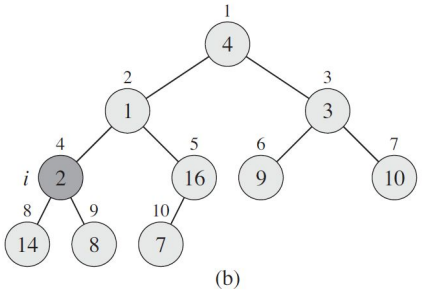
na sredini je posljednji čvor s djecom

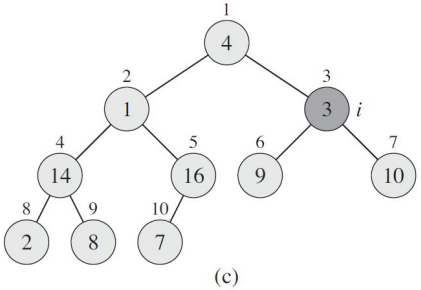
- kreće od listova stabla (n/2) i ide prema gore
- za svaki čvor poziva MAKS-GOMILANJE

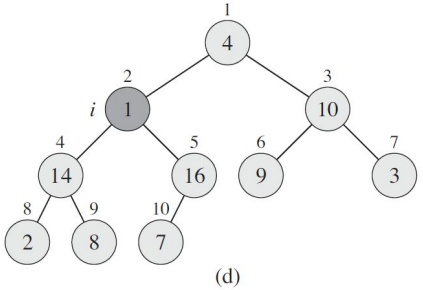


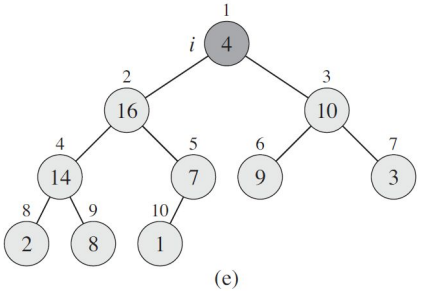
Slika 3.1: Binarna gomila kao (a) stablo i (b) niz.

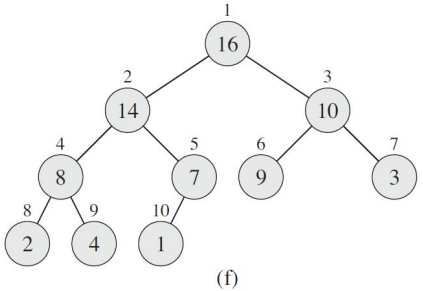






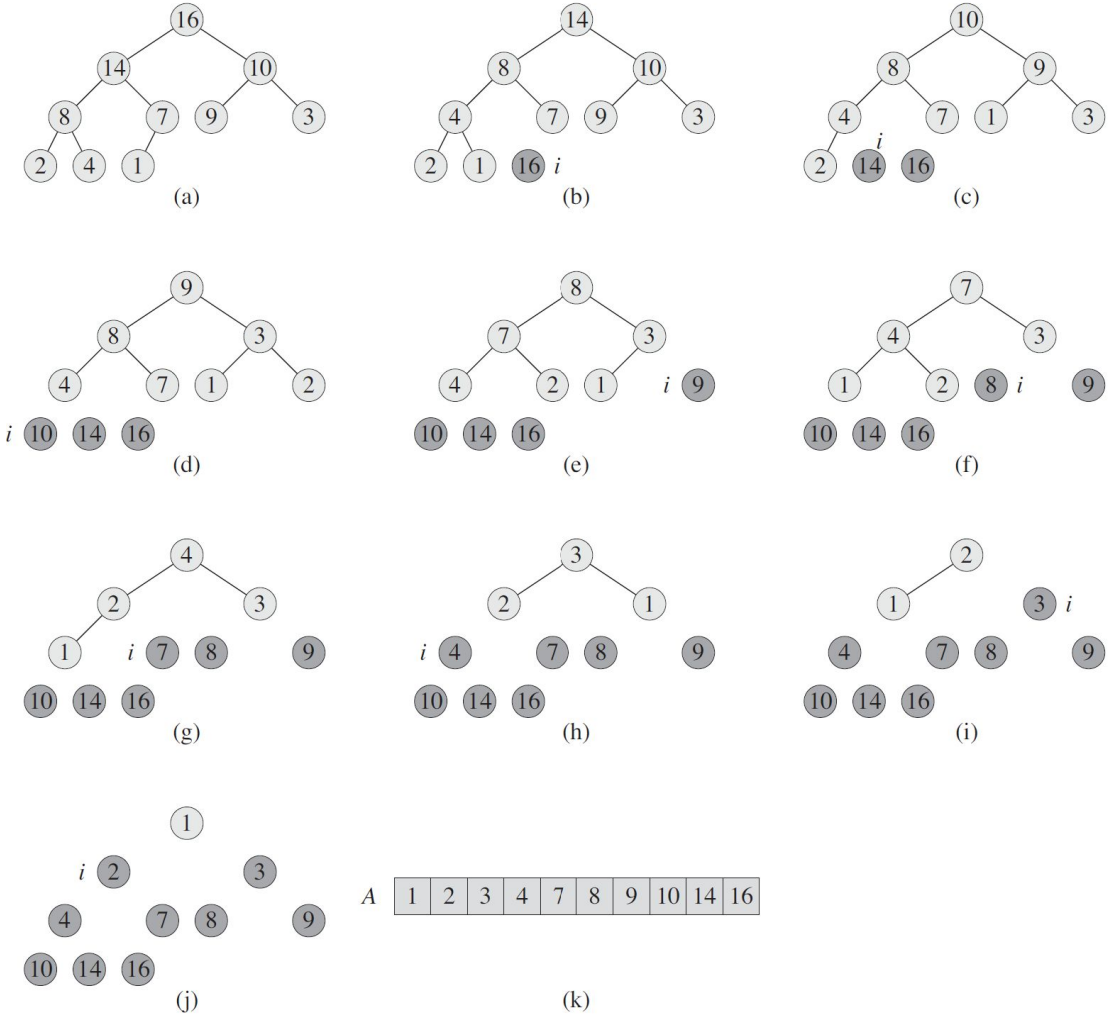






Sortiranje: algoritam HeapSort

- Radi u O(n lg n) i sortira niz u mjestu, nestabilan
- Prvo zamijeni korijen s posljednjim elementom i smanji veličinu gomile za 1
- Nakon toga pozove MAKS-GOMILANJE nad prvim elementom
- Ovo ponavlja za sve preostale čvorove u gomili



Sortiranje: algoritam HeapSort

Algoritam 3.13 Algoritam *Heapsort* [7]. Parametar A je ulazni niz, a n veličina gomile.

```
1: procedure HEAPSORT(A, n)
2:     IZGRADI-MAKS-GOMILU(A, n)
3:     for i ← n downto 2 do
4:         zamijeni A[1] i A[i]
5:         A.velicina_gomile ← A.velicina_gomile − 1
6:         MAKS-GOMILANJE(A, 1)
7:     end for
8: end procedure
```

Prioritetni red

- Gomila je efikasna struktura podataka za *prioritetni red*
- Prioritetni red je skup elemenata gdje svaki od njih ima pridruženu brojčanu vrijednost koja se zove *ključ*
- Ključ određuje prioritet (ili “važnost”) elementa
- Zavisno od vrste gomile, prioritetni red može biti *maks-prioritetni-red* ili *min-prioritetni-red* (implementacija je slična kao i kod maks/min-gomile)
- Prioritetni red ima primjenu u teoriji grafova, umjetnoj inteligenciji, kompresiji podataka, operacijskim sustavima i dr.

Operacije nad gomilom kao prioritetnim redom

- GOMILA-MAKSIMUM - vraća element s najvećim ključem
- GOMILA-IZDVOJI-MAKSIMUM - izdvaja element s najvećim ključem
- GOMILA-POVEĆAJ-KLJUČ - povećava ključ (prioritet) zadanog elementa
- GOMILA-DODAJ - dodaje novi element u gomilu
- Sve ove operacije rade u O(lg n) u najgorem slučaju i služe za implementaciju prioritetnog reda

Operacije prioritetnog reda: GOMILA-MAKSIMUM

- GOMILA-MAKSIMUM - vraća element s najvećim ključem

Algoritam 3.14 Funkcija koja vraća maksimum gomile. Parametar A je gomila kao ulazni niz.

```
1: function GOMILA-MAKSIMUM(A)
2:     if A.velicina_gomile < 1 then
3:         greška "gomila je prazna"
4:     end if
5:     return A[1]
6: end function
```

Operacije prioritetnog reda: GOMILA-IZDVOJI-MAKSIMUM

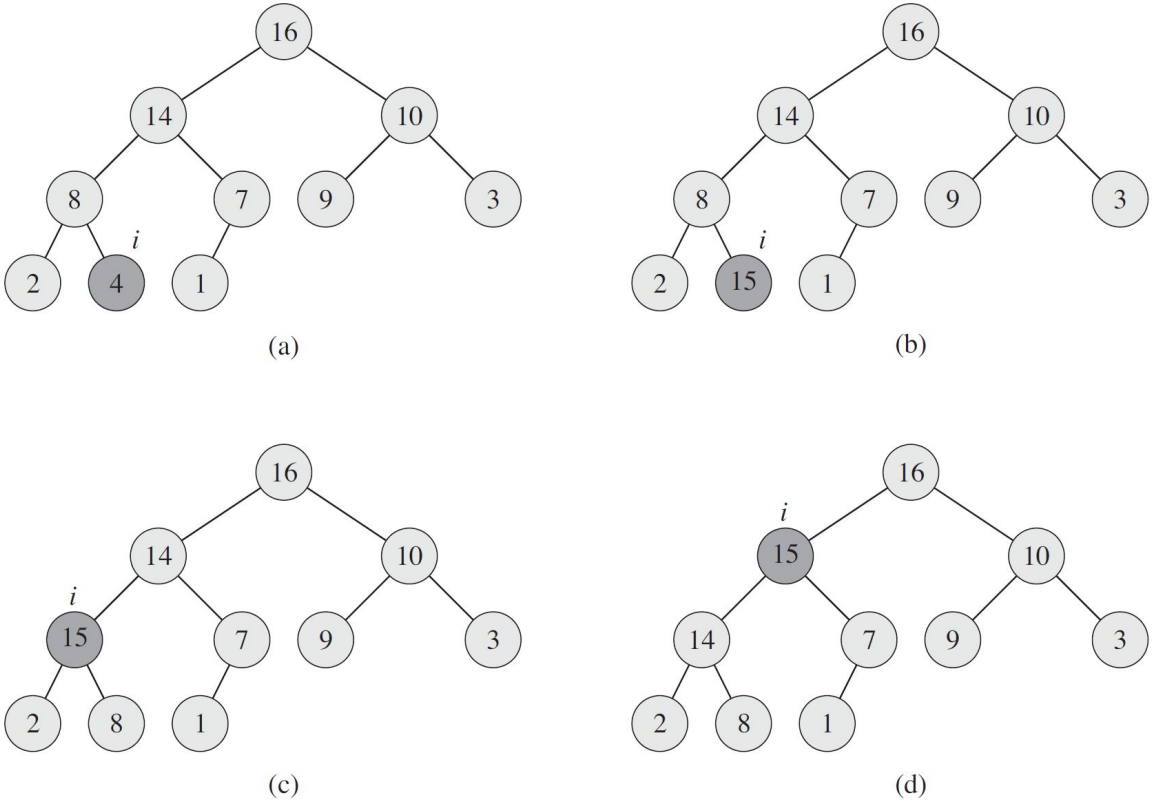
- GOMILA-IZDVOJI-MAKSIMUM - uklanja i vraća element s najvećim ključem

Algoritam 3.15 Funkcija koja uklanja i vraća maksimum gomile. Parametar A je gomila kao ulazni niz.

```
1:  function IZDVOJI-MAKSIMUM(A)
2:      if A.velicina_gomile < 1 then
3:          error "gomila je prazna"
4:      end if
5:      maks ← A[1]
6:      A[1] ← A[A.velicina_gomile]
7:      A.velicina_gomile ← A.velicina_gomile − 1
8:      MAKS-GOMILANJE(A, 1)
9:      return maks
10: end function
```

prenosi posljednji element na početak

Operacije prioritetnog reda: GOMILA-POVEĆAJ-KLJUČ



Operacije prioritetnog reda: GOMILA-POVEĆAJ-KLJUČ

- GOMILA-POVEĆAJ-KLJUČ - povećava vrijednosta ključa zadanog elementa na novu vrijednost koja nije manja od one trenutne

Algoritam 3.16 Operacija za povećavanje ključa [7]. Parametar A je ulazni niz, x je vrijednost čiji ključ povećavamo, a k novi ključ.

```
1:  procedure POVEĆAJ-KLJUČ(A, x, k)
2:      if k < x.kljuc then
3:          greška "novi ključ ne smije biti manji od postojećeg"
4:      end if
5:      x.kljuc ← k
6:      nađi index i na kojem se nalazi x
7:      while i > 1 and A[RODITELJ(i)].kljuc < A[i].kljuc do
8:          zamijeni A[i] sa A[RODITELJ(i)]
9:          i ← RODITELJ(i)
10:     end while
11: end procedure
```

zadan je element, ne indeks

- jer ne koristimo MAKS-GOMILANJE

Operacije prioritetnog reda: GOMILA-DODAJ

- GOMILA-DODAJ - dodaje novi element u gomilu

Algoritam 3.17 Dodavanje elementa u gomilu [7]. Parametar A je ulazni niz, x je novi element, a n veličina niza.

```
1:  procedure DODAJ(A, x, n)
2:      if A.velicina_gomile = n then
3:          greška "novi element ne stane u niz"
4:      end if
5:      A.velicina_gomile ← A.velicina_gomile + 1
6:      k ← x.kljuc
7:      x.kljuc = −∞
8:      A[A.velicina_gomile] = x
9:      POVEĆAJ-KLJUČ(A, x, k)
10: end procedure
```

mogli bi koristit MAKS-GOMILANJE, ali ovako je malo efikasnije

postavi ključ na -∞ da POVEĆAJ-KLJUČ postavi element na pravo mjesto

smjesti x na kraj gomile

Primjer

- Zadana je lista brojeva i cijeli broj k. Cilj je pronaći najvećih k elemenata u listi.
- Korištenje min-gomile omogućava nam da održavamo najveće elemente, kako dolaze

Primjer

- Postupak:
  1. Inicijaliziraj min-gomilu veličine k s prvih k elemenata iz liste
  2. Za svaki od preostalih elemenata u listi:
  * Ako je element veći od korijena gomile, zamijeni korijen ovim elementom.
  * Rebalansiraj gomilu (jer umetanje ili brisanje elementa u gomilu traje O(log k) vremena).
  3. Nakon obrade svih elemenata, min-gomila sadržavat će k najvećih elemenata u listi.
- Ovaj pristup ima vremensku složenost O(n lg k), što je bolje u usporedbi s O(n log n), koliko je potrebno za potpuno sortiranje liste.

Primjer

- [10,30,20,25,60,15,70,40] i želimo prva *k* = 3 najveća elementa

1. Inicijalizacija min-gomile s prvih *k* elemenata:
   Gomila = [10,30,20] (svojstvo min-gomile je zadovoljeno).
2. Sljedeći elementi
   * 25: 25 > 10 (korijen), zamijeni 10 s 25, min-gomilanje -> [20, 30, 25]
   * 60: 60 > 20, zamijeni 60 s 20, min-gomilanje -> [25, 30, 60]
   * 15: 15 < 25, nema promjene
   * 70: 70 > 25, zamijeni 70 s 25, min_gomilanje -> [30, 70, 60]
   * 40: 40 > 30, zamijeni 40 s 30, min_gomilanje -> [40, 70, 60]

- Konačan niz sadrži tri najveća elementa: [40, 70, 60]

Primjer – implementacija u Pythonu

```
import heapq

def k_najvecih(niz, k):
    # inicijaliziramo min-gomilu s prvih k elemenata
    min_gomila = niz[:k]
    heapq.heapify(min_gomila)

    # obrađujemo preostale elemente u listi
    for broj in niz[k:]:
        if broj > min_gomila[0]:  # ako je trenutni element veći od korijena …
            heapq.heapreplace(min_gomila, broj)

    return min_gomila  # vraćamo k najvećih elemenata

niz = [10, 30, 20, 25, 60, 15, 70, 40]
k = 3
print(k_najvecih(niz, k))  # ispisuje [40, 70, 60]
```

Sažetak

- Maks-gomila je struktura podataka koja je skoro potpuno binarno stablo kod kojeg je svaki čvor veći od svojih sljedbenika (ili manji, ako se radi o min-gomili)
- Gomila se može efikasno implementirati kao običan niz (nije potrebno implementirati stablo)
- Ova struktura podataka je temelj algoritma Heapsort čija je vremenska složenost $O(n \lg n)$ u najgorem slučaju
- Gomila se najčešće upotrebljava za efikasnu implementaciju prioritetnog reda
- Dvije osnovne operacije gomile su uspostavljanje svojstva gomile (MAKS-GOMILANJE) i stvaranje gomile iz niza (IZGRADI-MAKS-GOMILU)
- Sve ove operacije su $O(\log_2 n)$

Sažetak

- Možemo li koristiti “običan” sortiran niz kao red prioriteta?
- Naravno, ali onda nam neke operacije neće biti $O(\log_2 n)$ …
- … kao IZDVOJI-MAKSIMUM - izdvaja element s najvećim ključem,
  POVEĆAJ-KLJUČ - povećava ključ (prioritet) zadanog elementa,
  DODAJ - dodaje novi element u gomilu
- U ovim bi slučajevima morali nanovo sortirati niz ili proći sve elemente niza, dakle $O(n)$ u najboljem slučaju