Još sortiranja ...

Sortiranje hipom

Algoritnmi sortiranja složenosti $O(n)$

Predmet: Uvod u Algoritme 17 - ESI053
Studijski program: Primenjeno softversko inženjerstvo

DEPARTMAN ZA RAČUNARSTVO I AUTOMATIKU

DEPARTMAN ZA ENERGETIKU, ELEKTRONIKU I KOMUNIKACIJE

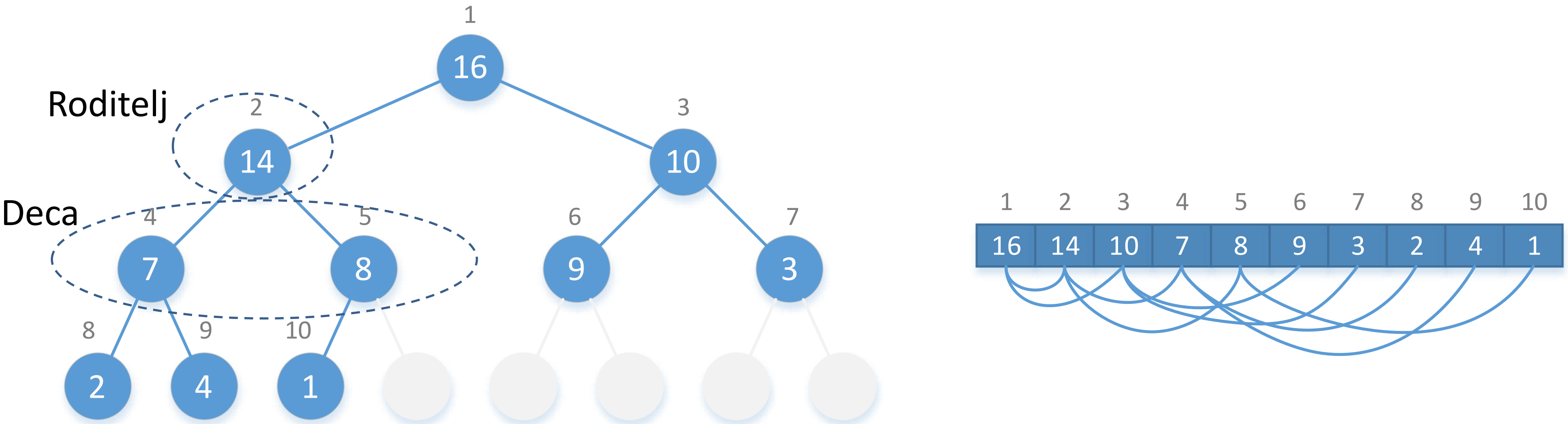
Sortiranje hipom - *Heapsort*

- Koristi se posebna “hip” (engl. *heap*) struktura podataka po kojoj je algoritam dobio ime (sa dugim „i“ u izgovoru)
  - Hip je pogodan za implementaciju efikasnog niza sa prioritetima (engl. *Priority Queue*)
- Složenost algoritma *heapsort* je $\Theta(n \cdot \log_2 n)$.
  - Brzina algoritma je odlična, ali je od njega bolji dobro implementiran *Quicksort* algoritam
- Algoritam radi u mestu jer se promene mesta elementima odvijaju u istom fizičkom nizu.

Hip struktura

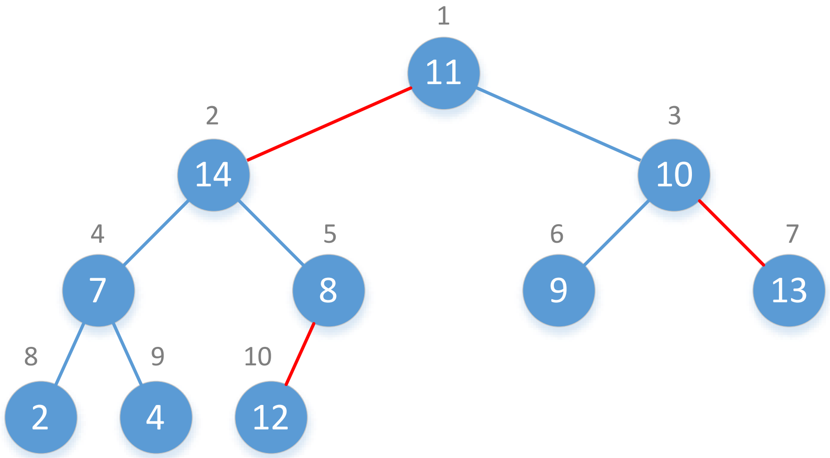
- Hip je niz elemenata koji se može predstaviti (logički) kao binarno stablo
  - Binarno stablo je skoro kompletno, tj. moguće je da poslednji nivo nije popunjen do kraja.
- Vrste *heap*-a
  - *Max-heap* (ima osobinu **A[Roditelj(i)]** ≥ **A[i]**), najveći elemenat je u korenu
  - *Min-heap* (ima osobinu **A[Roditelj(i)]** ≤ **A[i]**), majmanji elemenat je u korenu
- Nadalje će se posmatrati *max-heap*

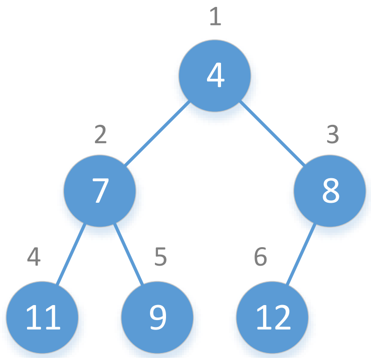
  Primer:



Primeri

- Levo – nije hip
- Desno – jeste *min heap*





1	2	3	4	5	6	7	8	9	10
11	14	10	7	8	9	13	2	4	12

1	2	3	4	5	6
4	7	8	11	9	12

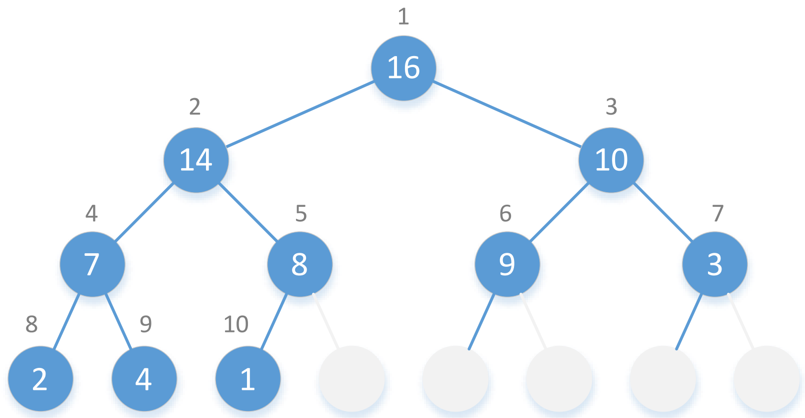
Osobine hip strukture

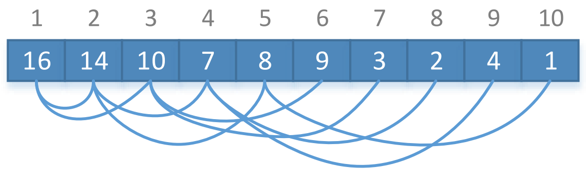
- Za *Heapsort* algoritam se koristi *max-heap*
- Visina hipa od n elemenata je broj logičkih nivoa $h = \log_2 n$
- Operacije sa *heap-om*:
  - Reorganizuj-Hip (Max-Heapify) održava *max-heap* osobinu (složenost $O(\log_2 n)$)
  - Izgradi-Hip (*Build-Max-Heap*) pravi *max-heap* na osnovu nesortiranog ulaznog niza (složenost $O(n)$)
  - Sortiraj-Hipom (*Heapsort*) sortira niz u mestu (složenost $O(n \log_2 n)$)
  - Hip struktura je pogodna za realizaciju reda sa prioritetima (složenost operacija $O(\log_2 n)$)

RODITELJ(i)
return $\lfloor i/2 \rfloor$

LEVO-DETE(i)
return $2i$

DESNO-DETE(i)
return $2i+1$





Reorganizuj-Hip metoda

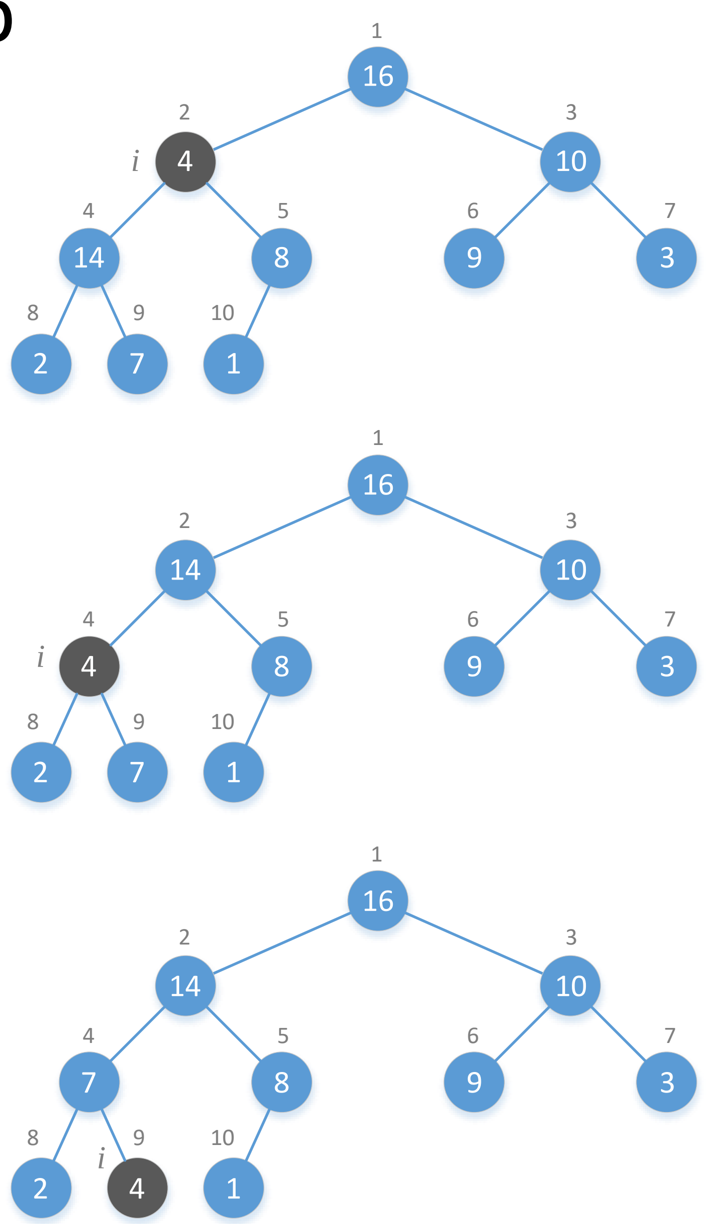
- Primenjuje se za izgradnju *Heap-a*
- Ako je u korenu podstabla vrednost manja nego u „deci“, proslediti tu vrednost „na dole“ tako da se održi osobina *Heap-a*.

```
REORGANIZUJ-HIP(A, i)
l = LEVO-DETE(i)
d = DESNO-DETE(i)
if l ≤ A.veličina-hipa and A[l] > A[i]
  najveći = l
else najveći = i
if d ≤ A.veličina-hipa and A[d] > A[najveći]
  najveći = d
if najveći ≠ i
  A[i] ↔ A[najveći]
  REORGANIZUJ-HIP(A, najveći)
```

Primer Reorganizuj-Hip

- Promene koje sprovodi REORGANIZUJ-HIP(A, 2)
  - primenjuju se na čvor 2
  - ako se vrednost zameni sa detetom i ono se reorganizuje (rekurzivno se promene primenjuju u dubinu)

```
REORGANIZUJ-HIP(A, i)
  l = LEVO-DETE(i)
  d = DESNO-DETE(i)
  if l ≤ A.vel-hipa & A[l] > A[i]
      najveći = l
  else najveći = i
  if d ≤ A.vel-hipa & A[d] > A[najveći]
      najveći = d
  if najveći ≠ i
      A[i] ↔ A[najveći]
      REORGANIZUJ-HIP(A, najveći)
```



Vreme izvršavanja REORGANIZUJ-HIP

- Zamena vrednosti u korenu sa nekim od dece je $\Theta(1)$ operacija, ali se vrednost iz korena može propagirati u dubinu rekurzivnim pozivima REORGANIZUJ-HIP
  - Za podstablo od n elemenata maksimalna veličina grane (podstabla sa jedne strane) je $2n/3$ (najgori slučaj je kada je poslednji nivo popunjen do pola – vidi prethodni primer: leva grana 6, a desna 3 elementa)

- Trajanje rekurzivnog poziva je

$$T(n) \leq T(2n/3) + \Theta(1)$$

$$T(n) = O(\log_2 n)$$

- Takođe, ovo vreme se može iskazati preko dubine (visine) hipa h

$$T(n) = O(h)$$

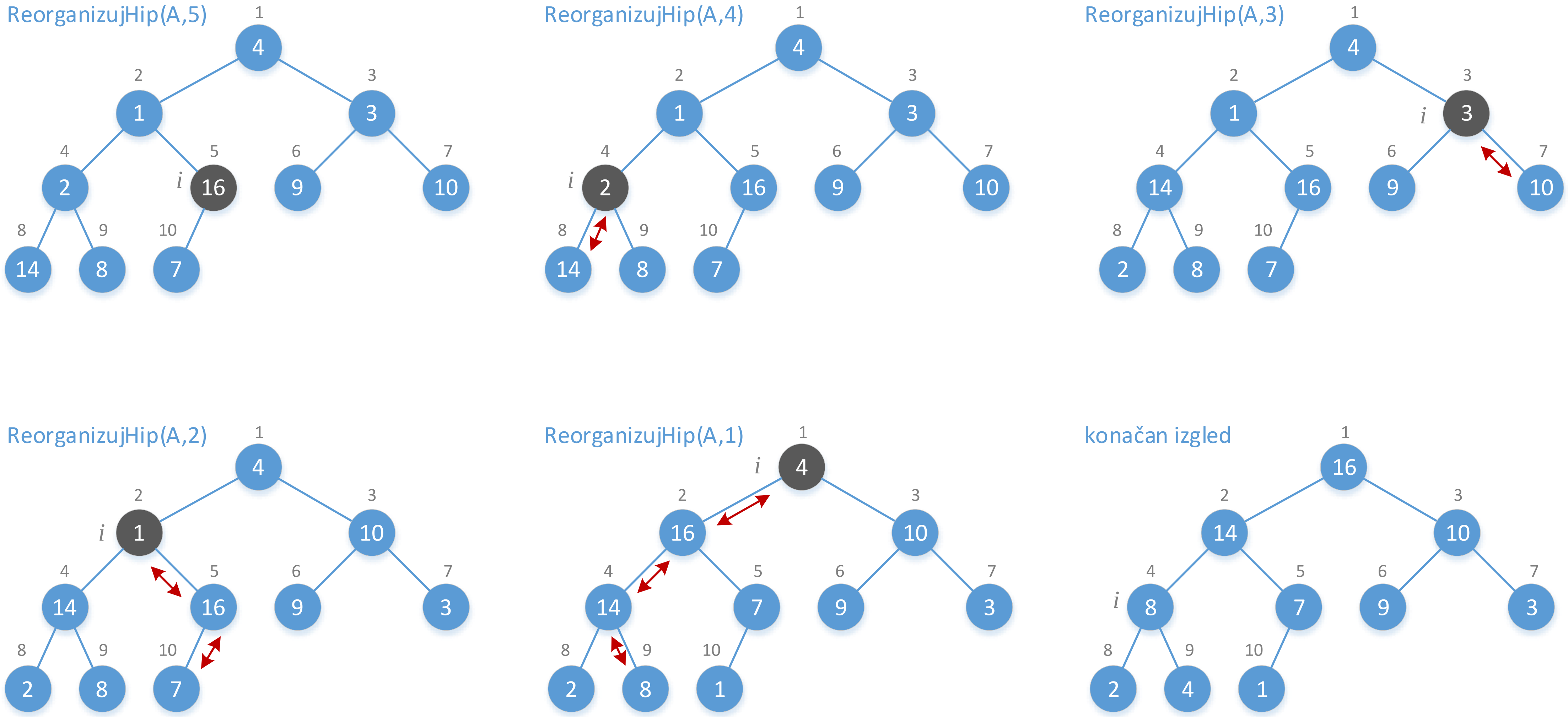
Izgradi-Hip

- Izgradnja hipa na osnovu niza A[1..n]
- Koristi se Reorganizuj-Hip metoda tako što se primeni (unazad) na svim elementima hipa koji nisu lišće (imaju dete)
  - To su elementi $A[1..\lfloor n/2 \rfloor]$, a preostali elementi su lišće $A[(\lfloor n/2 \rfloor + 1)..n]$

```
IZGRADI-HIP(A)
A.veličina-hipa = A.length
for i = ⌊A.length/2⌋ downto 1
    REORGANIZUJ-HIP(A, i)
```

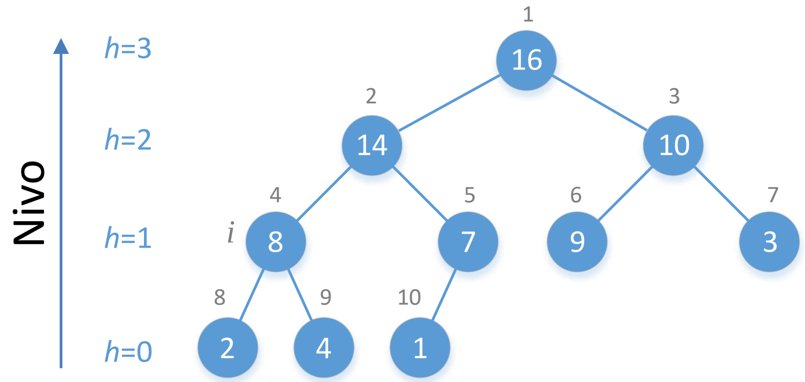
Primer izgradnje hipa

	1	2	3	4	5	6	7	8	9	10
A:	4	1	3	2	16	9	10	14	8	7



Vreme izvršavanja izgradnje hipa

- Posmatramo hip od n elemenata (n je veliko)
  - Dubina hipa je $\lfloor \log_2 n \rfloor$
  - Broj elemenata u jednom nivou h je $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
  - Trajanje jednog poziva Reorganizuj-Hip na nivou h je $O(h)$



- Ukupno vreme izvršavanja je $O(n)$
  Dokaz:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) \leq O\left(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

jer je

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}, \quad |x| < 1$$

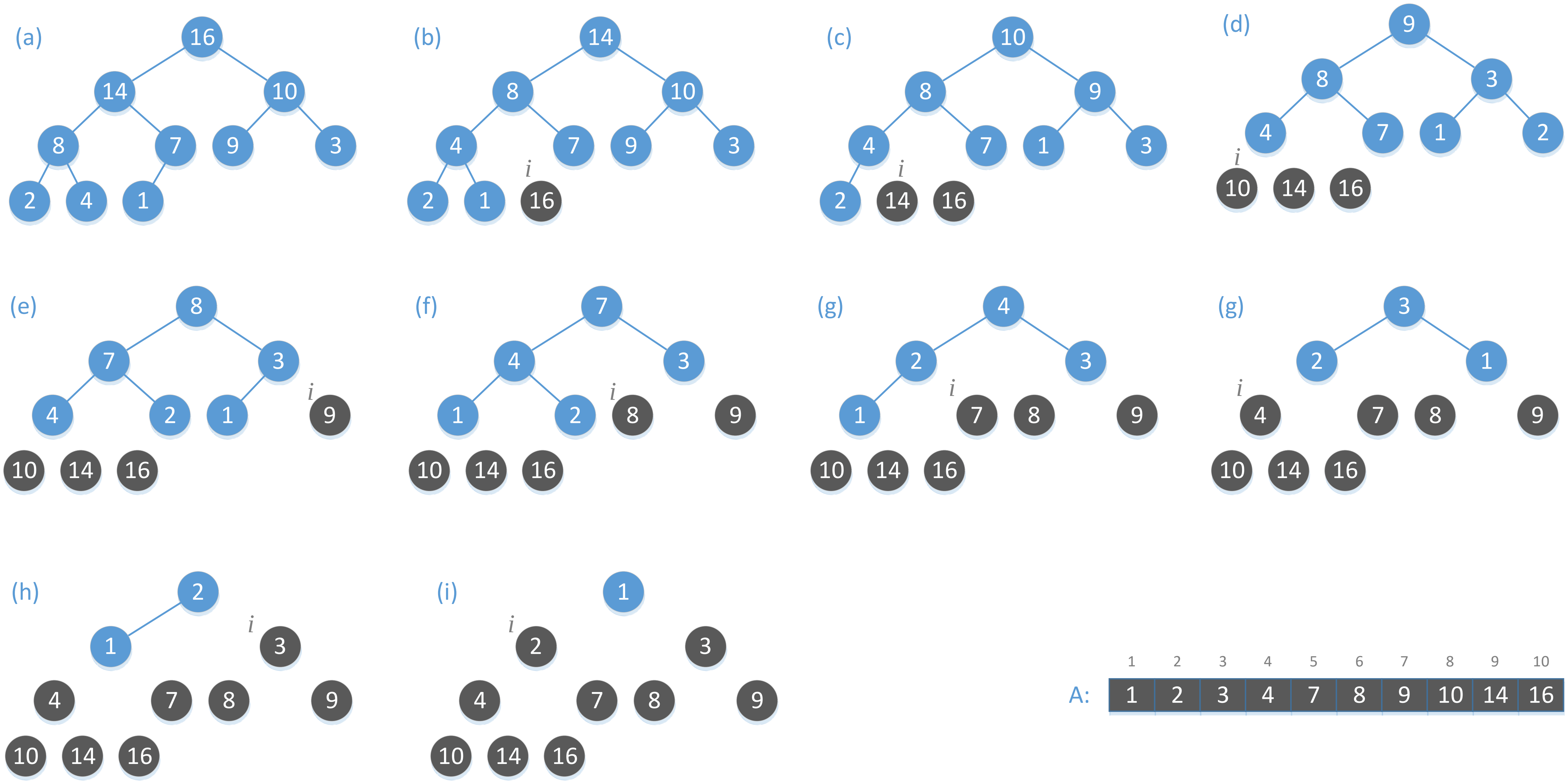
$$\sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

Algoritam sortiranja hipom - *Heapsort*

- Započinje izgradnjom hipa
- Nadalje, najveći elemenat u nizu (koji je u korenu hipa) se zameni sa poslednjim elementom niza A, niz se skrati za 1 elemenat i koriguje se poredak (poziv Reorganizuj-Hip(A,1))
- Nastavlja sa prethodnim korakom dok ima elemenata u nizu A

```
SORTIRANJE-HIPOM(A)
  IZGRADI-HIP(A)
  for i = A.length downto 2
      A[1] ↔ A[i]
      A.veličina-hipa = A.veličina-hipa - 1
      REORGANIZUJ-HIP(A, 1)
```

Primer sortiranja hipom



	1	2	3	4	5	6	7	8	9	10
A:	1	2	3	4	7	8	9	10	14	16

Složenost sortiranja hipom

- Izgradnja hipa je $O(n)$
- Nadalje se dešava $n - 1$ poziv Reorganizuj-Hip čija je složenost $O(\log_2 n)$
  - Tj. $T(n) = O(n) + (n-1)O(\log_2 n) = O(n \log_2 n) + O(n) - O(\log_2 n) = O(n \log_2 n)$

- Sledi, složenost sortiranja hipom je $O(n \cdot \log_2 n)$

Red sa prioritetima (*Priority Queue*)

- struktura podataka *Priority Queue* organizuje skup podataka *S* gde svaki elemenat ima udružen prioritet – kao „ključ“.
- *Max Priority Queue* podržava operacije:
  - DODAJ(S, X) dodaje elemenat x u skup S ($S = S \quad \{x\}$) (INSERT).
  - MAKSIMUM(S) vraća element iz S sa najvećim ključem (MAXIMUM).
  - IZDVOJ-MAKSIMUM(S) uklanja i vraća element iz S sa najvećim ključem (EXTRACT-MAX).
  - POVEĆAJ-KLJUČ(S,x,k) povećava vrednost (ključa) elementa x na k (pod uslovom da je $k >$ tekućeg ključa) (INCREASE-KEY).
- Primer upotrebe: raspoređivač zadataka u operativnom sistemu u redu sa prioritetima čuva zadatke spremne na izvršenje.

- Slično, *Min Priority Queue* podržava operacije:
  - DODAJ, MINIMUM, IZDVOJ-MINIMUM, SMANJI-KLJUČ

Implementacija reda sa prioritetima

- Upotrebljen je *max heap*

Složenost: $O(1)$

```
MAKSIMUM(A)
  return A[1]
```

Složenost: $O(\log_2 n)$

```
IZDVOJ-MAKSIMUM(A)
  if A.veličina-hipa < 1 error
  max = A[1]
  A[1] = A[A.veličina-hipa]
  A.veličina-hipa=A.veličina-hipa-1
  REORGANIZUJ-HIP(A, 1)
  return max
```

Složenost: $O(\log_2 n)$

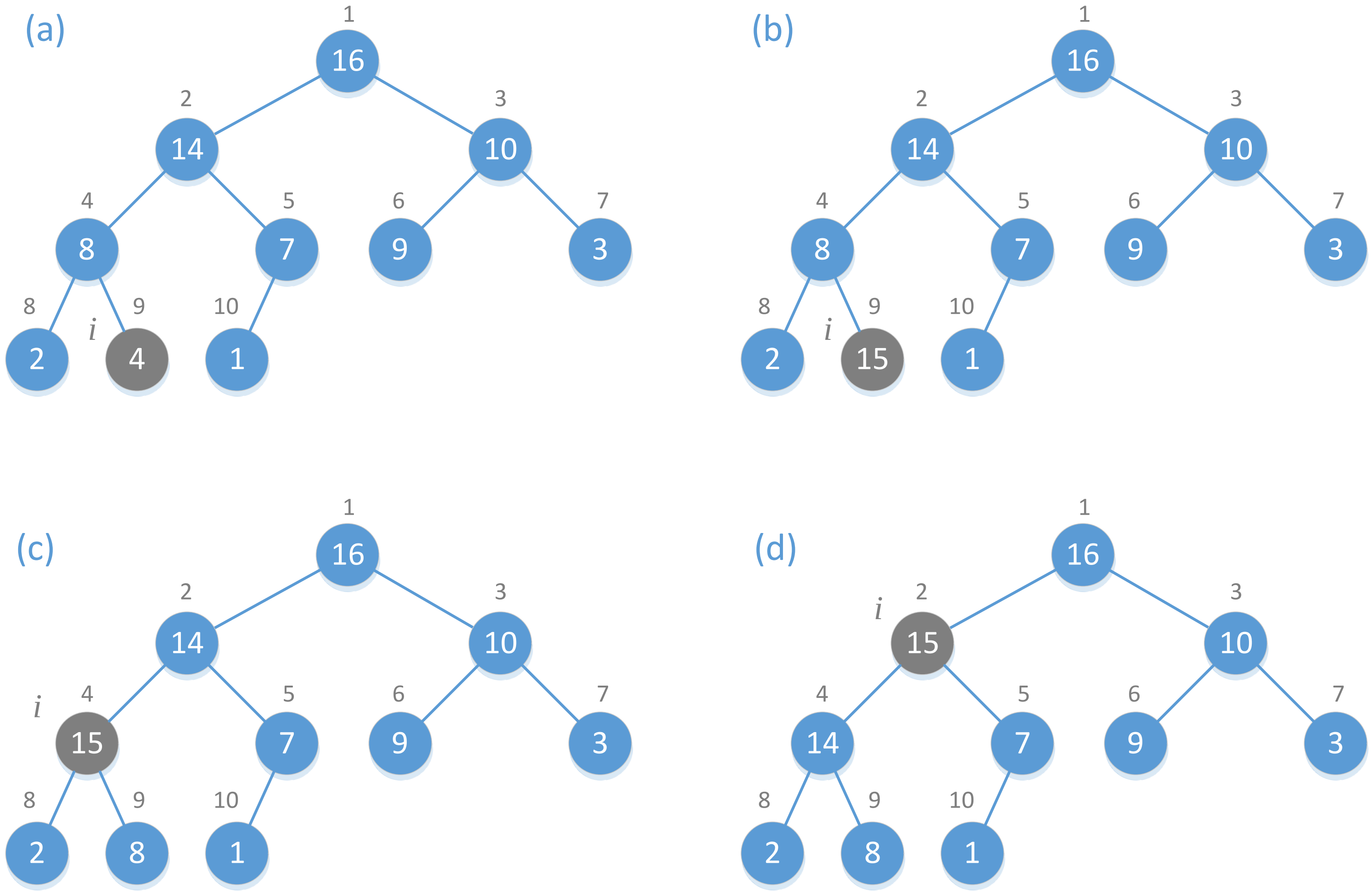
```
POVEĆAJ-KLJUČ(A, i, ključ)
  if ključ < A[i] error
  A[i] = ključ
  while i > 1 and A[RODITELJ(i)] < A[i]
     A[RODITELJ(i)] ↔ A[i]
     i = RODITELJ(i)
```

Složenost: $O(\log_2 n)$

```
DODAJ(A, ključ)
  A.veličina-hipa=A.veličina-hipa+1
  A[A.veličina-hipa] = -∞
  POVEĆAJ-KLJUČ(A,A.veličina-hipa,ključ)
```

Primer *Priority Queue*

- Prioritet zadatka #9 (*i*=9) je promenjen sa 4 na 15 …

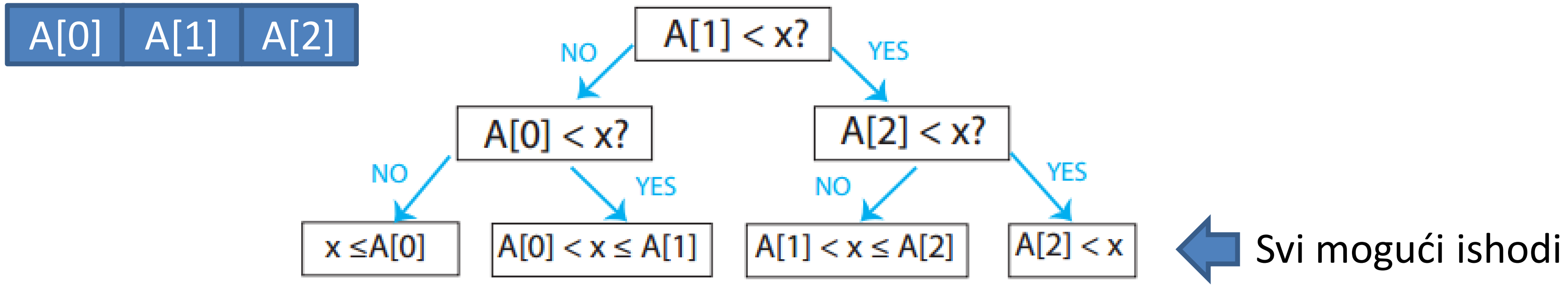


Model poređenja - *Comparison model*

- Model računanja koji se oslanja na poređenje se može primeniti na proizvoljan tip podataka
  - Tj. elementi koje posmatramo su apstraktni tipovi podataka (ADTs – *Abstract Data Types*)
  - Postoje operacije poređenja elemenata (<, >, ≤, …)
- Trajanje algoritma (u modelu poređenja) se iskaže **brojem operacija poređenja**

Stablo odlučivanja (*Decision Tree*)

- Svaki algoritam gde se upotrebljava model poređenja može se prikazati kao stablo sa svim mogućim ishodima poređenja (za dato n)
- Primer: Binarna pretraga ($n = 3$)

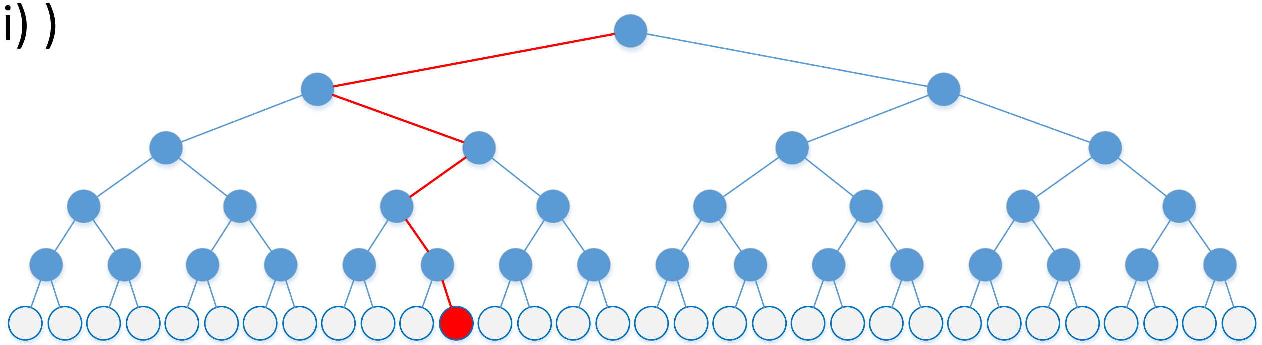


- Stablo odlučivanja
  - **Unutrašnji čvorovi** (na svim nivoma sem donjeg) imaju binarne odluke: 0-1 ili Da-Ne (na slici Yes-No)
  - **List** (na donjem nivou) je izlaz (algoritam je gotov)
  - **Putanja** od korena (gonji čvor) do lista predstavlja izvršavanje algoritma
  - **Dužina putanje** (dubina) odgovara vremenu izvršavanja
  - **Visina stabla** je najgori slučaj izvršavanja algoritma (predstavlja najdužu putanju)

Donja granica brzine pretrage

Pretpostavke:

- Svih n elemenata je pretprocesirano (npr. sortirano)
- Broj listova ≥ broja mogućih odgovora ≥ n
  - (barem jedan ishod za svaki elemenat A(i))
- Stablo odlučivanja je binarno stablo
- Visina stabla ≥ $\log_2 n$



Zaključak:

- Problem pretrage u modelu poređenja je složenosti $\Omega(\log_2 n)$
- Binarna pretraga je optimalna (kada se posmatra u modelu poređenja)

Donja granica brzine sortiranja

Pretpostavke:

- Svaki list je moguća permutacija elemenata niza
  - Npr. A[3] < A[15] < A[1] < …
- Broj listova je jednak broju permutacija = $n!$
- Visina stabla ≥ $\log_2 n!$

$$= \log_2(1{\cdot}2{\cdot}3 \ldots {\cdot}(n-1){\cdot}n) = \log_2 1 + \log_2 2 + \cdots + \log_2 n$$

$$= \sum_{i=1}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2} (\log_2 n - 1) = \Omega(n \log_2 n)$$

Zaključak:

- Sortiranje u modelu poređenja je složenosti $\Omega(n \log_2 n)$

Sortiranje složenosti $O(n)$

- Dosadašnji algoritmi sortiranja su koristili međusobna poređenja vrednosti
  - Model poređenja pokazuje da je najmanja složenost $\Omega(n \log_2 n)$!!!

- Da li moguće realizovati brži algoritam? DA
  - Npr., gde vreme trajanja linearno raste sa veličinom niza!
- *Counting sort*, *Bucket sort* i *Radix sort* su primeri takvih algoritama

U čemu je „trik“? Kako je moguće da su brži od teorijske granice?

- Odgovor je u upotrebi dodatnih informacija, a ne samo oslanjanje na poređenje vrednosti
  - Npr. *Counting sort* zahteva da su ključevi celi brojevi u manjem opsegu (fizički raspoložive memorije)

Sortiranje prebrojavanjem - *Counting sort*

- Algoritam podrazumeva da se sortiraju brojevi koji su u opsegu $0..k$
- Princip: za svaki element x, algoritam izbroji koliko ima brojeva manjih od x
  - Primer: ako postoji 17 brojeva manjih od x, onda se x nalazi na 18. poziciji u sortiranom nizu
- Implementacija algoritma ima dva dodatna niza
  - $B[1..n]$ je niz sortiranih brojeva
  - $C[0..k]$ je privremeni niz

- Osobine:
  - Složenost algoritma ja $\Theta(n+k)$
  - Takođe, memorijsko zauzeće je $\Theta(n+k)$
  - Algoritan je **stabilan** (*stable*) – brojevi iste vrednosti se u izlaznom nizu pojavljuju u istom poretku kao što su u ulaznom nizu

Counting sort - algoritam

Algoritam ima 3 dela – koraka:

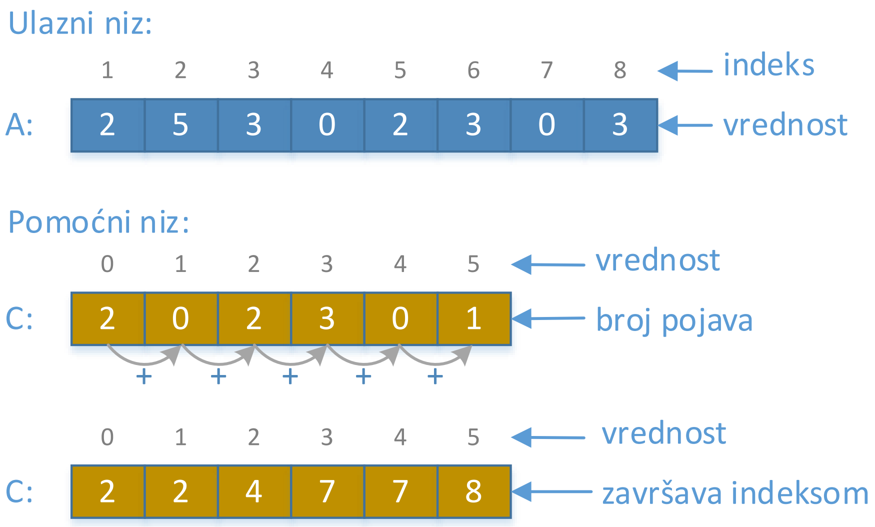
1. prebroji pojave iste vrednosti i zapiše u niz C
2. napravi kumulativnu sumu pojava da dobije pozicije gde će se naći poslednje pojave iste vrednosti
3. rasporedi vrednosti u novi niz B

Korak 3 izgleda suvišno, ali nije kada postoje satelitski podaci.

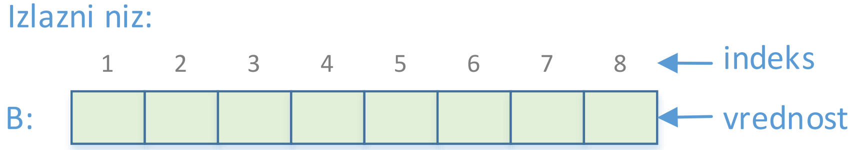
Napomena: podrazumeva se da su vrednosti u opsegu $0..k$

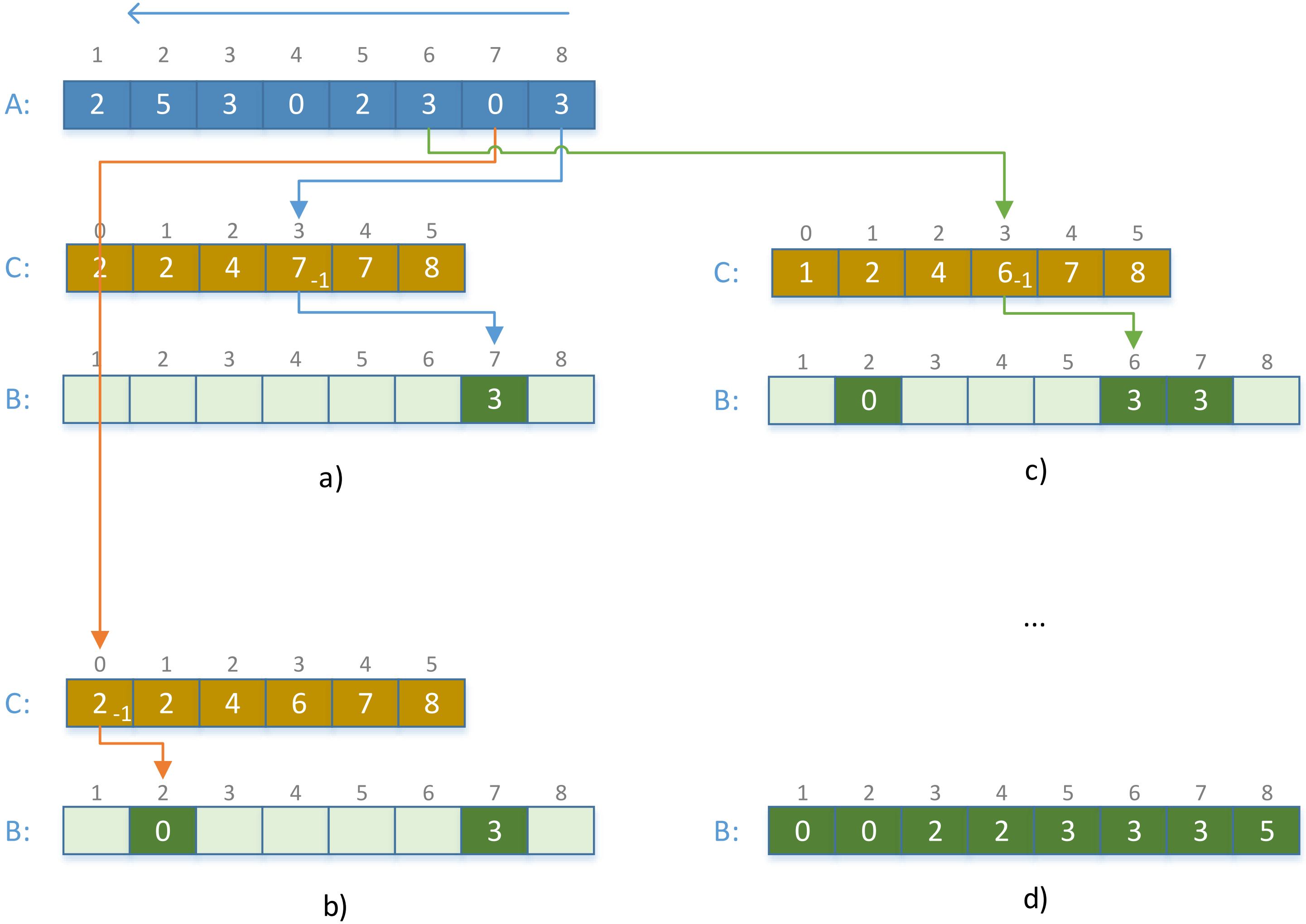
```
SORTIRANJE-PREBROJAVANJEM(A, B, k)
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
for i = 1 to k
    C[i] = C[i] + C[i-1]
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Counting sort - Primer



Npr., vrednost 3 završava indeksom 7, a vrednost 2 završava indeksom 4. Znači ima 3=7-4 pojave vrednosti 3 koje se nalaze na indeksima 5,6,7





Radiks sortiranje - *Radix sort*

- Npr. posmatramo broj sačinjen od d cifara
- Radix sort koristi *Counting sort* algoritam da sortira cifru-po-cifru polazeći od najniže cifre.
  - Ovo je moguće jer je *Counting sort* stabilan algoritam.

- Primer:

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

- Algoritam

```
SORTIRANJE-RADIX(A, d)
  for cifra = 1 to d          // najniža cifra je 1
     Sortiraj cifru i stabilnim sort alg.
```

Radix sort ili *Quicksort*?

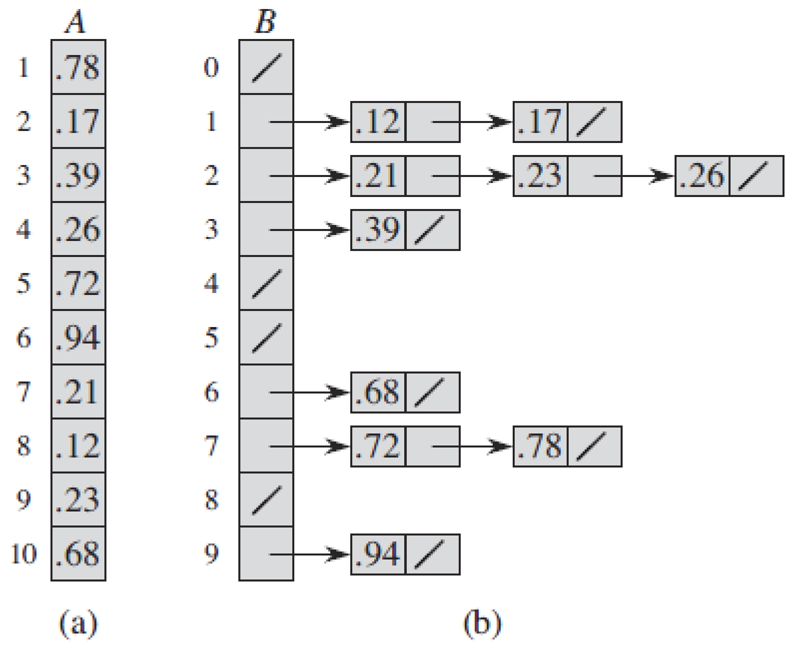
- Složenost *Radix sort*-a:
  - Broj cifara $d = \log_b k$ i svaka cifra $\in \{0, 1, \dots, b-1\}$
  - Za svaku cifru $\Theta(n + b)$
  - Ukupno $\Theta\big((n+b)d\big) = \Theta\big((n+b)\log_b k\big)$
  - Minimalna složenost je za $n = b$: $\Theta(n \log_n k) = O(nc)$ za $k \le n^c$

- Složenost *Radix sort*-a $\Theta(n)$, a *Quicksort*-a je $\Theta(n \log_2 n)$
  - $\Theta(n)$ je bolje od $\Theta(n \log_2 n)$, ali ...
  - Konstantan faktor *Radix sort*-a je lošiji od *Quicksort*-a
  - Dobre implementacije *Quicksort*-a su brže od *Radix sort*-a

Segmentno sortiranje - *Bucket sort*

- Kod niza čije vrednosti imaju uniformnu raspodelu u intervalu [0,1) srednje vreme izvršavanja *Bucket sort*-a je $O(n)$
- interval vrednosti se deli na $m = \Theta(n)$ podintervala jednakih veličina
  - u primeru je interval vrednosti [0,1), a broj podintervala je 10

- Princip:
  - ulazni niz A ima elemente $0 \leq A[i] < 1$
  - niz B[0..*n*-1] sadrži podnizove („kofe“) iz A asocirane odgovarajućim podintervalima
  - Svaki podniz se sortira
    - primenom nekog drugog sort alg. Ili
    - Rekurzivnom primenom ovog alg.



Bucket sort - algoritam

```
SORTIRANJE-SEGMENTNO(A, B)
n = A.length
for i = 0 to n-1
    B[i] = ∅
for i = 1 to n
    ubaciti A[i] u grupu B[⌊n·A[i]⌋]
for i = 0 to n-1
    Sortirati grupu B[i]    ← npr.Insertion sort
spojiti grupe B[0], B[1], … B[n-1]
```