Algoritmi

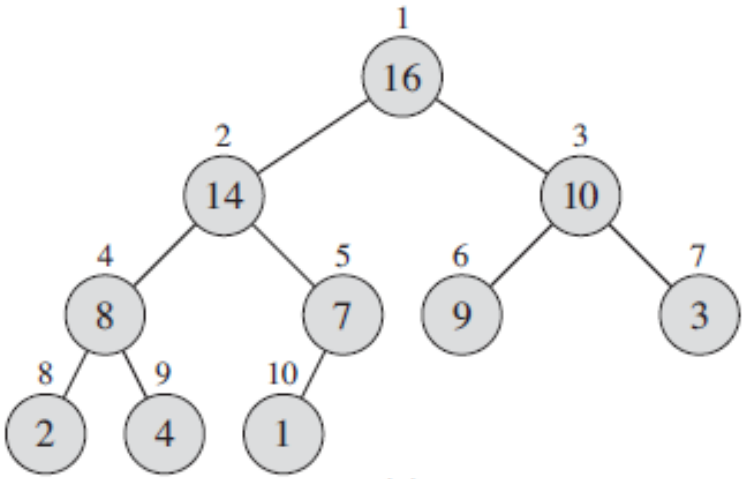
Sortiranje (nastavak)

Heapsort

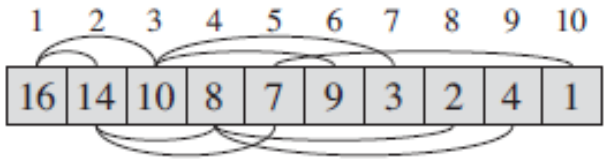
- Koristi se posebna struktura podataka *heap* (“hip”), po kojoj je algoritam dobio ime
  - *Heap* je pogodan za implementaciju efikasnog niza sa prioritetima (*Priority Queue*)
- Složenost algoritma *heapsort* je $n \cdot \log_2 n$.
  - Brzina algoritma je odlična, ali je od njega bolji dobro implementiran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

Heap struktura

- *Heap* je niz elemenata koji se može predstaviti kao binarno stablo
  - Binarno stablo je skoro kompletno, tj. moguće je da poslednji nivo nije popunjen do kraja.
- Vrste *heap*-a
  - *Max-heap* (A[Parent(i)] $\geq$ A[i]), najveći elemenat je u korenu
  - *Min-heap* (A[Parent(i)] $\leq$ A[i]), najmanji elemenat je u korenu
- Primer pravilno popunjenog *max-heap*-a



(a)



(b)

Osobine *heap* strukture

- Za *Heapsort* algoritam se koristi *max-heap*
- Visina *heap*-a je broj nivoa
  - Za n elemenata visina iznosi $\log_2 n$
- Operacije sa *heap*-om:
  - **MAX-HEAPIFY** održava max-heap osobinu
  - **BUILD-MAX-HEAP** pravi *max-heap* na osnovu nesortiranog ulaznog niza
  - **HEAPSORT** sortiran niz u mestu
  - **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, **HEAP-MAXIMUM** služe za implementaciju priority queue-a

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i+1
```

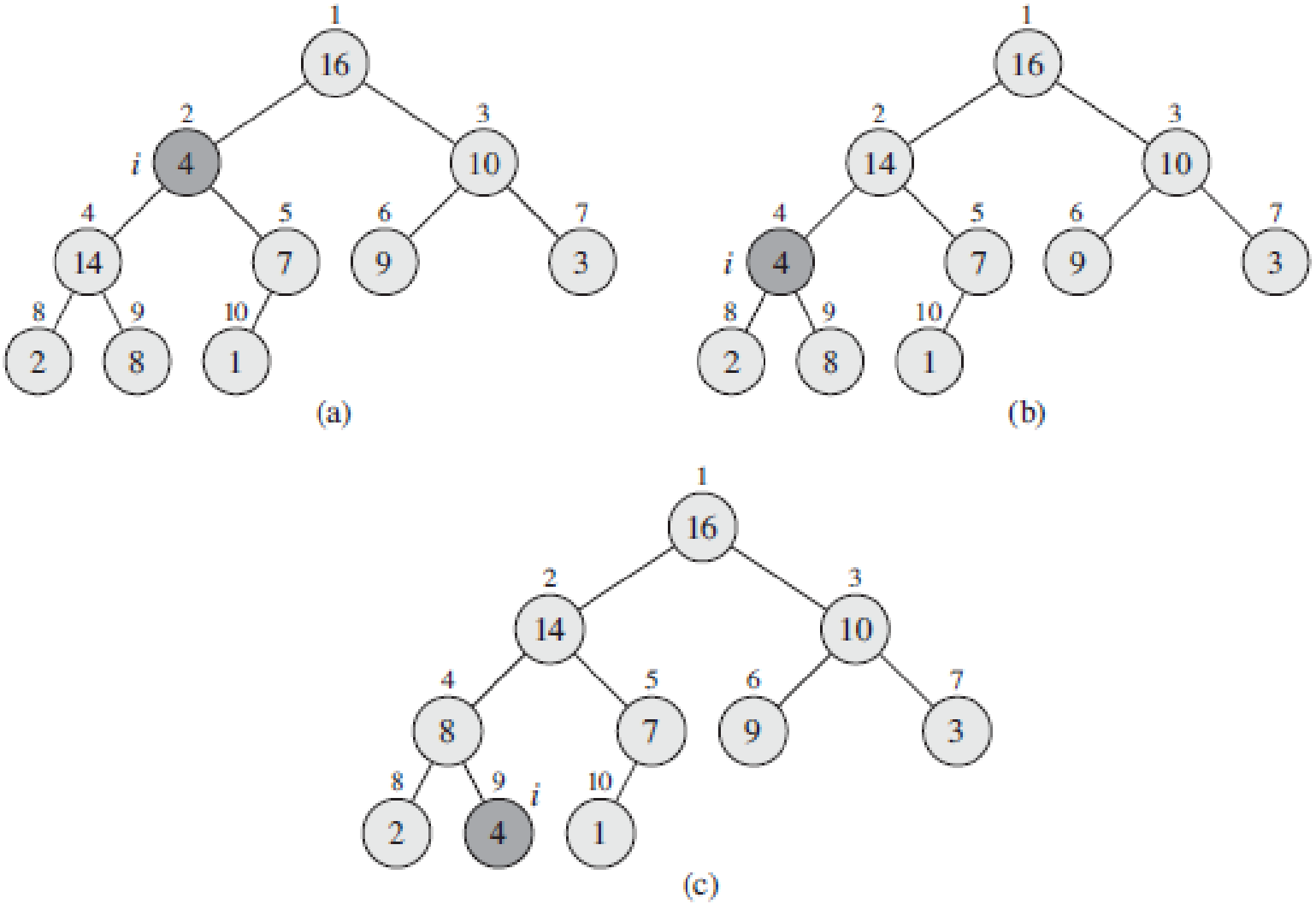
MAX-HEAPIFY metoda

- 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.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4    largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7    largest = r
8  if largest ≠ i
9    A[i] ↔ A[largest]
10   MAX-HEAPIFY(A, largest)
```

Primer MAX-HEAPIFY

- Promene koje sprovodi MAX-HEAPIFY(A,2)



Vreme izvršavanja MAX-HEAPIFY

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

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

- Trajanje rekurzivnog poziva je

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

- Takođe, ovo vreme se može iskazati preko dubine stabla *h*

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

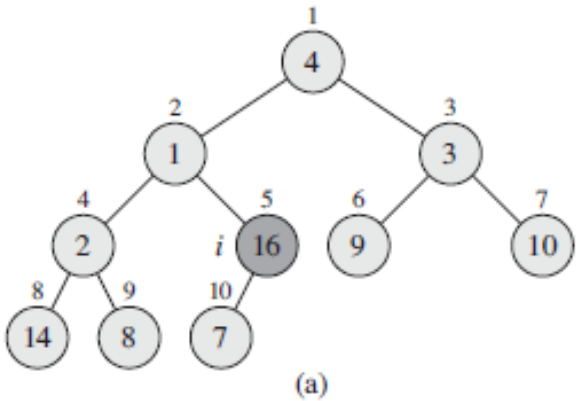
BUILD-MAX-HEAP

- Izgradnja *Max-Heap*-a na osnovu niza A[1..*n*]
- Koristi se MAX-HEAPIFY metoda tako što se primeni (unazad) na svim elementima Heap-a koji nisu lišće (elementi A[1..$\lfloor n/2 \rfloor$] , a preostali elementi su lišće A[($\lfloor n/2 \rfloor$+1)..*n*])

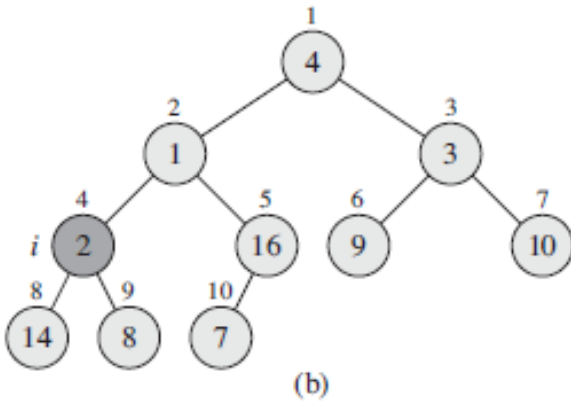
```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3     MAX-HEAPIFY(A, i)
```

Primer

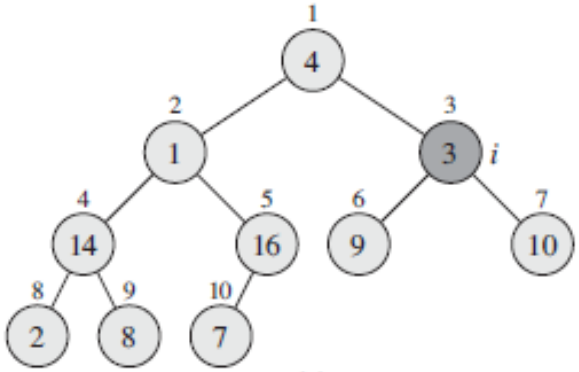
BUILD-MAX-HEAP



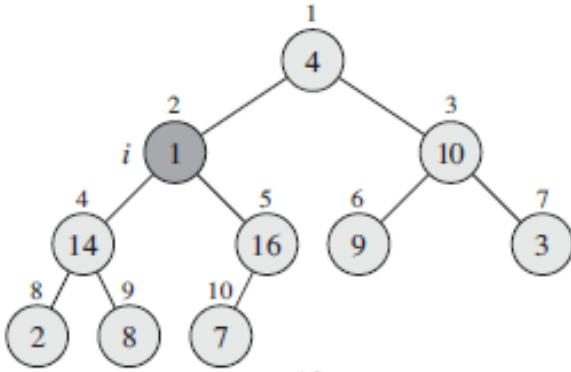
(a)



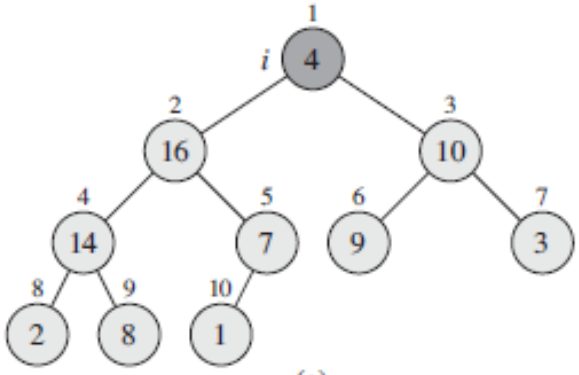
(b)



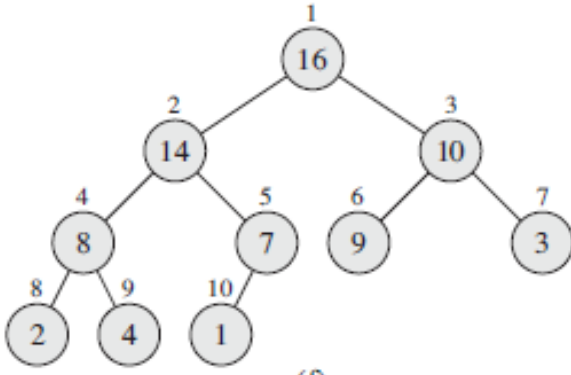
(c)



(d)



(e)



(f)

Vreme izvršavanja BUILD-MAX-HEAP

- Posmatramo heap od *n* elemenata
  - Dubina je $\lfloor \log_2 n \rfloor$
  - Broj elemenata u jednom nivou *h* je $\lceil n/2^{h+1} \rceil$
  - Trajanje jednog poziva Max-Heapify na nivou *h* je $O(h)$

- Ukupno vreme izvršavanja je $O(n)$

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

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

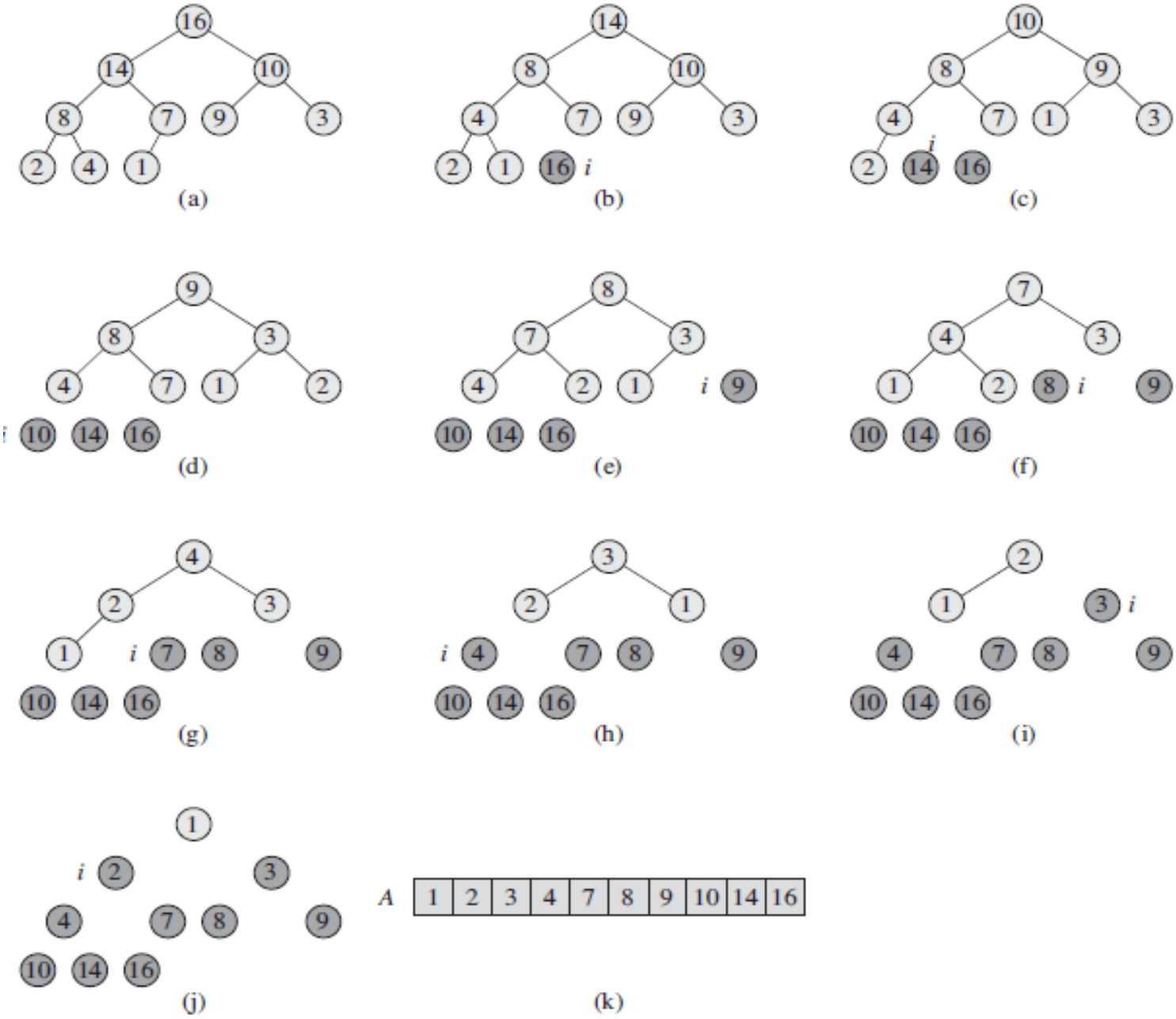
$$\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

Heapsort algoritam

- Započinje Build-Max-Heap metodom
- Nadalje, najveći elemenat u nizu (koji je u korenu *heap*-a) zamenjuje sa poslednjim elementom niza A, skraćuje niz za 1 elemenat i koriguje poredak (poziv Max-Heapify(A,1))
- Nastavlja sa prethodnim korakom dok ima elemenata u nizu A

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3     A[1] ↔ A[i]
4     A.heap-size = A.heap-size - 1
5     MAX-HEAPIFY(A, 1)
```

Primer *Heapsort*



Vreme izvršavanja *Heapsort*

- Build-Max-Heap je $O(n)$
- Nadalje se dešava n-1 poziv Max-Heapify koji je $O(\log_2 n)$

- Vreme izvršavanja *Heapsort*-a je $O(n \cdot \log_2 n)$

$$T(n) = O(n) + (n-1) \cdot O(\log_2 n) = O(n \log_2 n) + O(n) - O(\log_2 n) = O(n \log_2 n)$$

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:
  - INSERT(S, x) dodaje elemenat *x* u skup *S* ($S = S \cup \{x\}$).
  - MAXIMUM(S) vraća element iz *S* sa najvećim ključem.
  - EXTRACT-MAX(S) uklanja i vraća element iz *S* sa najvećim ključem.
  - INCREASE-KEY(S,x,k) povećava vrednost (ključa) elementa x na k (pod uslovom da je k > tekućeg ključa).
- Primer upotrebe: raspoređivač zadataka u operativnom sistemu u *Max Priority Queue* čuva zadatke spremne na izvršenje.

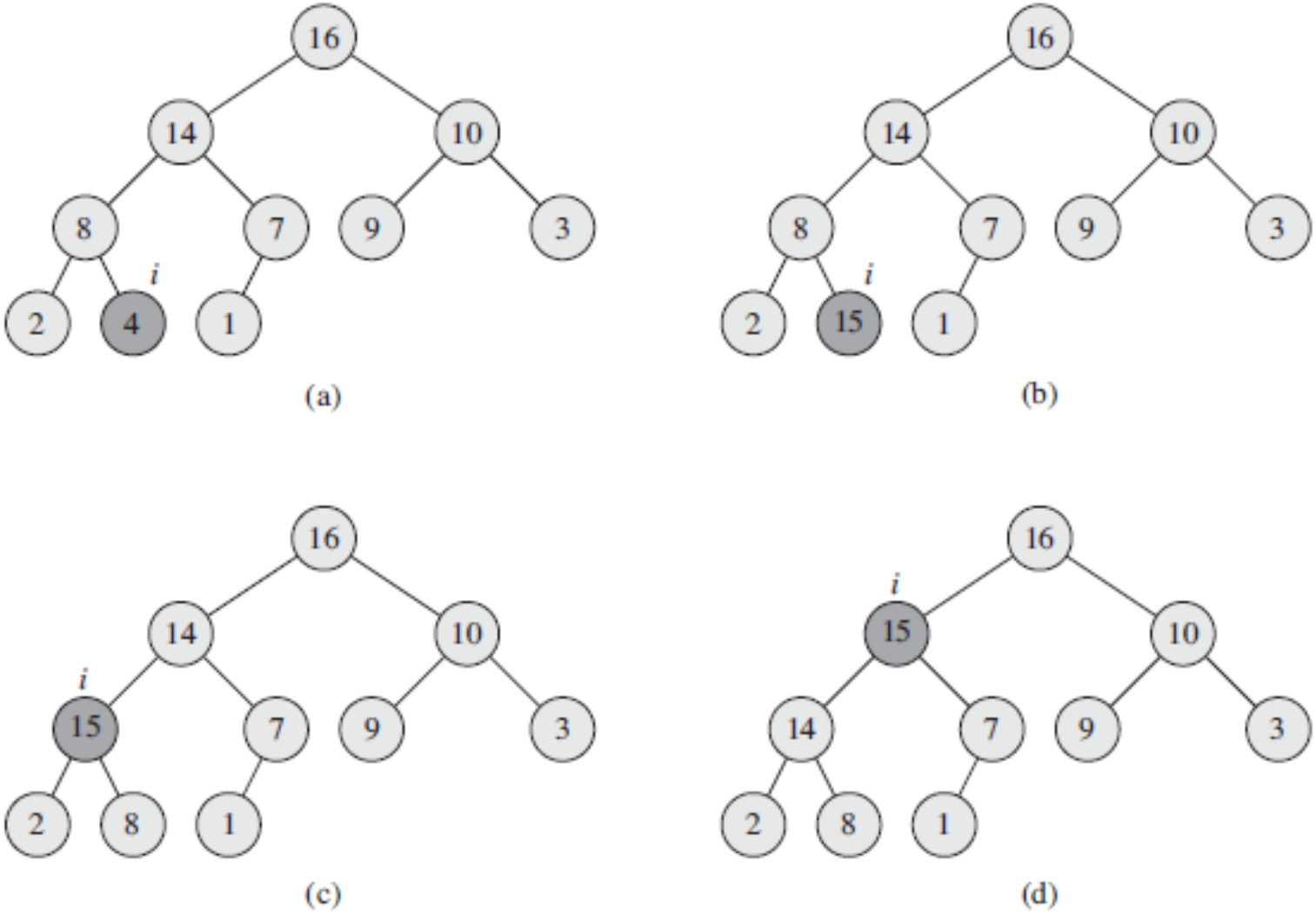
- Slično, *Min Priority Queue* podržava operacije:
  - INSERT, MINIMUM, EXTRACT-MIN, DECREASE-KEY

Implementacija *Priority Queue*

```
HEAP-MAXIMUM(A)
1  return A[1]

HEAP-EXTRACT-MAX(A)
1  if A.heap-size<1 error
2  max = A[1]
3  A[1] = A[A.heap-size]
4  A.heap-size=A.heap-size-1
5  MAX-HEAPIFY(A, 1)
6  return max

HEAP-INCREASE-KEY(A,i,key)
1  if key<A[i] error
2  A[i]=key
3  while i>1 and A[PARENT(i)]<A[i]
4     A[PARENT(i)] ↔ A[i]
5     i=PARENT(i)

MAX-HEAP-INSERT(A,key)
1  A.heap-size=A.heap-size+1
2  A[A.heap-size]=-∞
3  HEAP-INCREASE-KEY(A,A.heap-size,key)
```

Primer *Priority Queue*

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

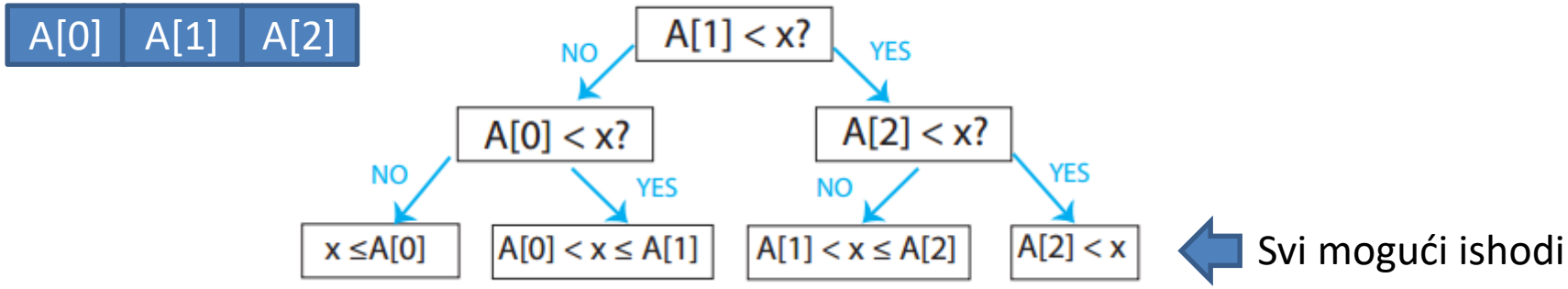


Model računanja – poređenje
“Comparison model”

- Elementi su apstraktni tipovi podataka (ADTs – *Abstract Data Types*)
- Postoji operacija poređenja elemenata (<, >, ≤, …)
- Trajanje algoritma 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)



  - Unutrašnji čvor = binarna odluka
  - List = izlaz (algoritam je gotov)
  - Putanja od korena do lista = izvršavanje algoritma
  - Dužina putanje (dubina) = vreme izvršavanja
  - Visina stabla = najgori slučaj izvršavanja alg.

Donja granica brzine pretrage

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

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

Donja granica brzine sortiranja

- List čini moguća permutacija elemenata niza
  - Npr. A[3] < A[15] < A[1] < …
- Broj listova je broj permutacija = $n!$
- Visina stabla ≥ $\log_2 n!$

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

$$= \sum_{i=1}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 \frac{n}{2} =$$

$$= \frac{n}{2}(\log_2 n - 1) = \underline{\Omega(n \log_2 n)}$$

Sortiranje složenosti $O(n)$

- Dosadašnji algoritmi sortiranja su koristili međusobna poređenja vrednosti

- Da li moguće realizovati brži algoritam? DA!
  - Gde vreme trajanja linearno raste sa veličinom niza!

- Counting sort, Bucket sort i Radix sort su primeri takvih algoritama

Counting sort

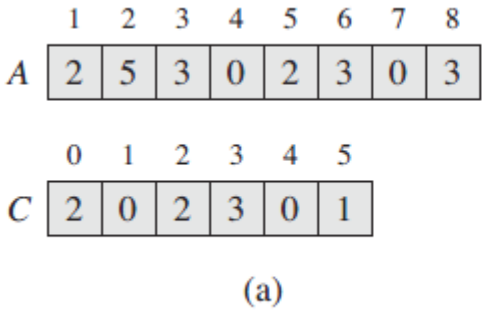
- Algoritam podrazumeva da se softiraju brojevi koji su u opsegu $0..k$
- Princip: za svaki element x, algoritam izbroji koliko ima brojeva manjih od x
  - Primer: ako postoji 17 brojeva manjih od x, onda se x nalazi na 18. poziciji u sortiranom nizu
- Implementacija algoritma ima dva dodatna niza
 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

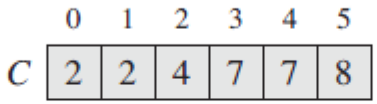
Counting sort - algoritam

```
COUNTING-SORT(A, B, k)
1  for i = 0 to k
2    C[i] = 0
3  for j = 1 to A.length
4    C[A[j]] = C[A[j]] + 1
5  for i = 1 to k
6    C[i] = C[i] + C[i-1]
7  for j = A.length downto 1
8    B[C[A[j]]] = A[j]
9    C[A[j]] = C[A[j]] - 1
```

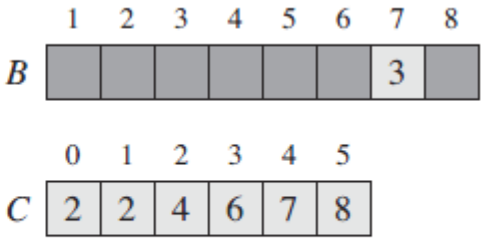
Counting sort - Primer



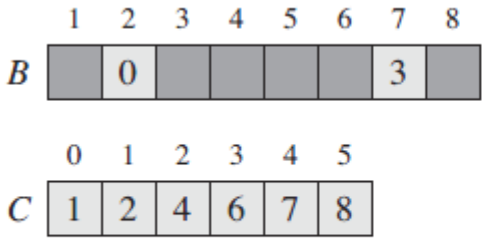
(a)



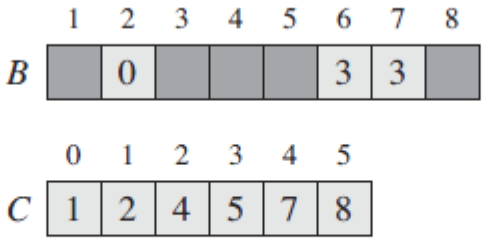
(b)



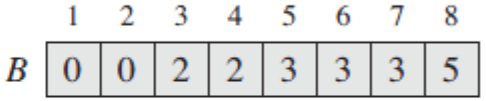
(c)



(d)



(e)

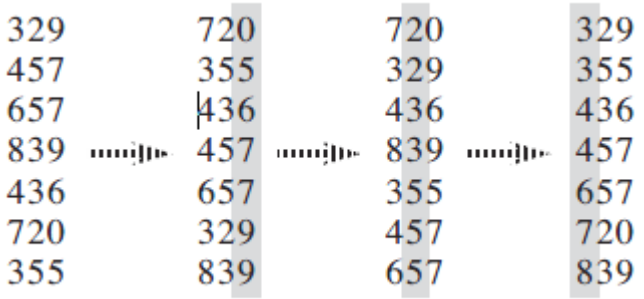


(f)

Radix sort

- Npr. posmatramo broj sačinjen od d cifara
- Radix sort koristi Counting algoritam da sortira cifra-po-cifra polazeći od najniže cifre

- Primer:



- Algoritam

```
RADIX-SORT(A, d)
1  for i = 1 to d
2     Sortiraj cifru i stabilnim sort alg.
```

Radix sort ili *Quicksort*?

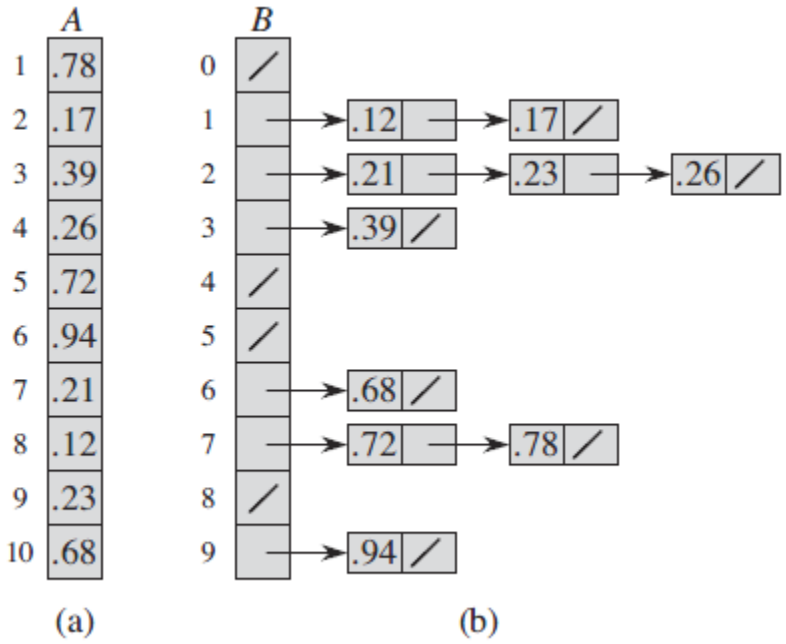
- Složenost *Radix sort*-a:
  - Broj cifara $d = \log_b k \ \in \{0, 1, \dots, b-1\}$
  - Za svaku cifru $\Theta(n+b)$
  - Ukupno $\Theta\big((n+b)d\big) = \Theta\big((n+b)\log_b k\big)$
  - Minimalna vrednost je za $n = b$: $\Theta(n \log_n k) = O(nc)$ za $k \leq n^c$

- 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

Bucket sort

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

- Princip:
  - ulazni niz A ima elemente $0 \leq A[i] < 1$
  - niz B[0..*n*-1] sadrži podnizove iz asocirane odgovarajućim podintervalima



Bucket sort - algoritam

```
BUCKET-SORT(A, B, k)
1  n = A.length
2  for i = 0 to n-1
3     B[i] = ∅
4  for i = 1 to n
5     ubaciti A[i] u grupu B[⌊n·A[i]⌋]
6  for i = 0 to n-1
7     Sortirati grupu B[i]   ← Insertion sort
8  spojiti grupe B[0], B[1], … B[n-1]
```