Red sa prioritetom, heap, adaptivni RSP

Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2023.

Red sa prioritetom

- red sa prioritetom čuva kolekciju elemenata
- svaki element je par (ključ, vrednost)
- osnovne operacije:
  - add(k, x): dodaje element sa ključem k i vrednošču x
  - remove_min(): uklanja element sa najmanjim ključem
- dodatne operacije:
  - min(): vraća, ali ne uklanja, element sa najmanjim ključem
  - len(), is_empty()

Primer operacija nad redom sa prioritetom

operacija	rezultat	sadržaj reda
`P.add(5, A)`	–	[(5,A)]
`P.add(9, C)`	–	[(5,A), (9,C)]
`P.add(3, B)`	–	[(3,B), (5,A), (9,C)]
`P.add(7, D)`	–	[(3,B), (5,A), (7,D), (9,C)]
`P.min()`	(3,B)	[(3,B), (5,A), (7,D), (9,C)]
`P.remove_min()`	(3,B)	[(5,A), (7,D), (9,C)]
`P.remove_min()`	(5,A)	[(7,D), (9,C)]
`len(P)`	2	[(7,D), (9,C)]
`P.remove_min()`	(7,D)	[(9,C)]
`P.remove_min()`	(9,C)	[ ]
`P.is_empty()`	True	[ ]
`P.remove_min()`	greška	[ ]

Ključevi i relacija poretka

- ključevi mogu biti bilo kog tipa za koga je definisana relacija poretka
- elementi u redu mogu imati jednake ključeve – u tom slučaju se primenjuje FIFO princip
- relacija poretka
  - refleksivna: $x \leq x$
  - antisimetrična: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - tranzitivna: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Element RSP

```python
class PriorityQueueItem:
  def __init__(self, k, v):
    self.key = k
    self.value = v

  def __lt__(self, other):
    return self.key < other.key

  def __le__(self, other):
    return self.key <= other.key
```

Implementacija RSP

- implementacija sa **nesortiranom** listom
- add je $O(1)$ jer dodavanje možemo raditi na bilo kom kraju liste
- remove_min i min su $O(n)$ jer moramo tražiti najmanji ključ u listi

- implementacija sa **sortiranom** listom
- add je $O(n)$ jer moramo da nađemo pravo mesto za ubacivanje novog elementa
- remove_min i min su $O(1)$ jer je najmanji ključ uvek na početku

RSP sa nesortiranom listom

```
class UnsortedPriorityQueue:
  def __init__(self):
    self._data = SingleList()

  def add(self, key, value):
    newest = PriorityQueueItem(key, value)
    self._data.add_last(newest)

  def _find_min(self)
    if self.is_empty():
      raise Empty('Queue is empty')
    smallest = self._data.first()
    current = smallest.next
    while current is not None:
      if current.element < smallest.element:
        smallest = curr
      current = current.next
    return smallest

  def remove_min(self):
    p = self._find_min()
    item = self._data.delete(p)
    return (item.key, item.value)
```

RSP sa sortiranom listom

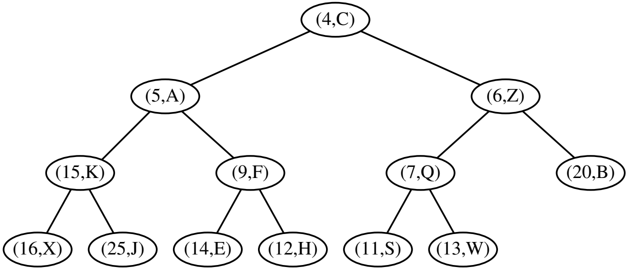
```
class SortedPriorityQueue:
  def __init__(self):
    self._data = SingleList()

  def add(self, key, value):
    newest = PriorityQueueItem(key, value)
    current = self._data.first()
    while current is not None and newest < current.element:
      current = current.next
    if current is None:
      self._data.add_first(newest)
    else:
      self._data.add_after(current, newest)

  def remove_min(self):
    if self.is_empty():
      raise Empty('Queue is empty')
    item = self._data.delete(self._data.first())
    return (item.key, item.value)
```

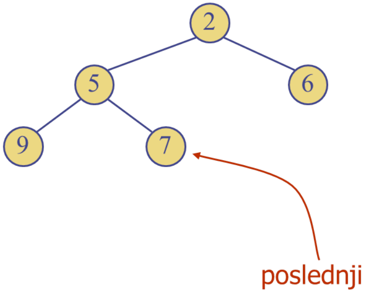
Heap

- heap je binarno stablo...
- ...čiji elementi su uređeni parovi (ključ, vrednost)
- ...i koje zadovoljava još 2 uslova:
  - **redosled**: za svaki čvor n osim korena ključ od n je veći ili jednak ključu roditelja od n
  - **kompletnost**: heap visine h ima nivoe $0, 1, 2, \ldots h-1$ sa maksimalnim brojem čvorova (i-ti nivo ima 2^i čvorova za $0 \leq i \leq h-1$)



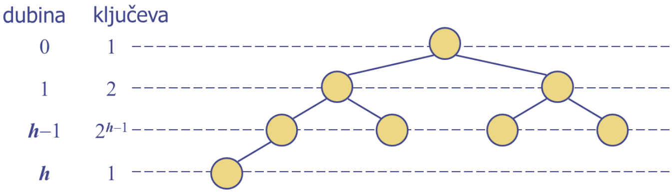
Heap

- poslednji čvor je poslednji čvor sa **desne** strane na najnižem nivou stabla



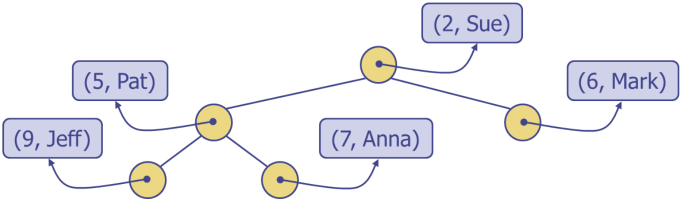
Dubina heapa

- **teorema:** heap koji čuva n ključeva ima dubinu $O(\log n)$
  - h: visina heapa sa n ključeva
  - ima 2^i ključeva na dubini $i = 0, \ldots h-1$ i bar jedan ključ na dubini h
  - prema tome, $n \geq 1 + 2 + 4 + \ldots 2^{h-1} + 1$
  - $n \geq 2^h$
  - $h \leq \log n$



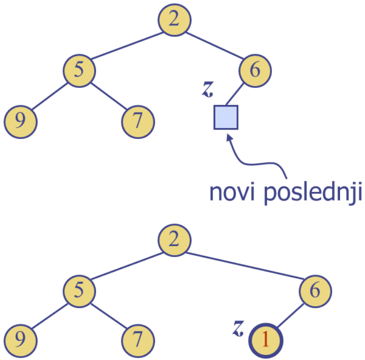
Heap i red sa prioritetom

- red sa prioritetom možemo implementirati pomoću heapa
- u svakom čvoru stabla čuvamo par (ključ, vrednost)
- pamtimo položaj poslednjeg čvora



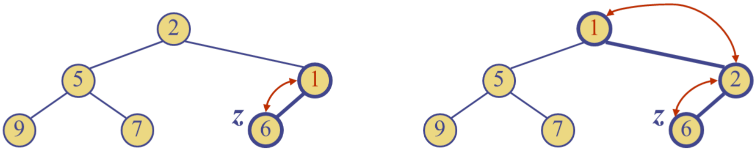
Dodavanje u heap

- add u redu sa prioritetom se implementira kao dodavanje u heap
- dodavanje se vrši u tri koraka
  1. nađi novi poslednji čvor z
  2. sačuvaj (k, v) u z
  3. restauriraj pravilan **redosled**



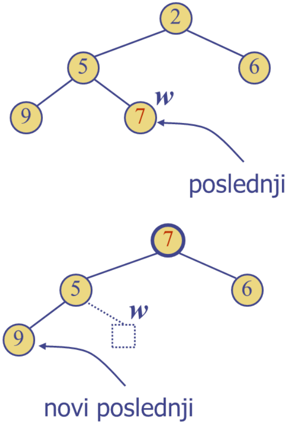
Dodavanje u heap: restauracija redosleda

- nakon dodavanja novog ključa k redosled čvorova može biti narušen
- algoritam upheap uspostavlja korektan redosled zamenom k duž putanje od novog čvora prema korenu
- **upheap** se završava kada k dođe u koren ili njegov roditelj ima ključ manji ili jednak k
- pošto heap ima visinu $O(\log n)$, upheap radi u $O(\log n)$ vremenu



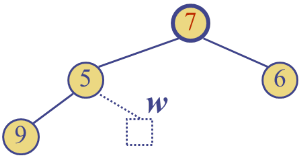
Uklanjanje iz heapa

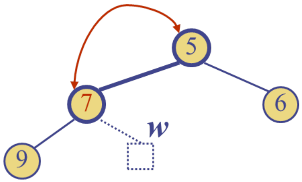
- remove_min se implementira kao uklanjanje korena iz heapa
- uklanjanje se vrši u tri koraka
    1. na mesto korena stavi poslednji čvor w
    2. ukloni w
    3. restauriraj pravilan **redosled**



Uklanjanje iz heapa: restauracija redosleda

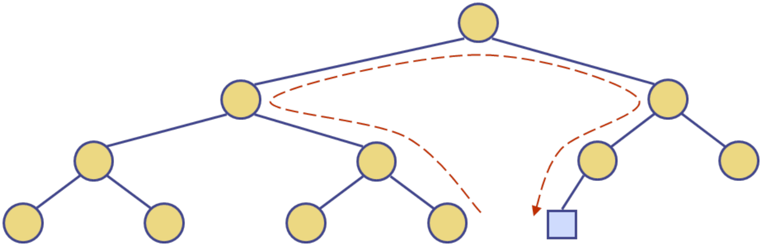
- nakon smeštanja ključa k poslednjeg čvora u koren redosled čvorova može biti narušen
- algoritam downheap uspostavlja korektan redosled zamenom k duž putanje od korena
- **downheap** se završava kada k dođe u list ili njegova deca imaju ključeve veće ili jednake k
- pošto heap ima visinu $O(\log n)$, downheap radi u $O(\log n)$ vremenu





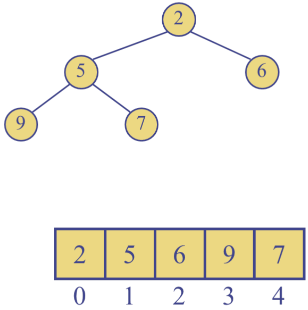
Nađi mesto za novi poslednji prilikom dodavanja

- mesto za novi poslednji čvor se može naći prolaskom kroz putanju od $O(\log n)$ čvorova
  - idi prema gore dok ne dođeš do korena ili nečijeg levog deteta
  - ako si došao do nečijeg levog deteta, idi na desno dete
  - idi prema dole levo dok ne dođeš do lista
- sličan je i algoritam prilikom uklanjanja



Implementacija heapa pomoću niza

- heap sa n ključeva se može smestiti u niz dužine n
- za čvor ranga i
  - levo dete ima rang $2i+1$
  - desno dete ima rang $2i+2$
- veze između čvorova se ne čuvaju
- dodavanje se svodi na upis čvora ranga $n+1$
- uklanjanje se svodi na uklanjanje čvora ranga n



Heap u Pythonu 1

```
class HeapPriorityQueue:
  def __init__(self):
    self._data = []

  def _parent(self, j):
    return (j-1)//2

  def _left(self, j):
    return 2*j+1

  def _right(self, j):
    return 2*j+2

  def _has_left(self, j):
    return self._left(j) < len(self._data)

  def _has_right(self, j):
    return self._right(j) < len(self._data)

  def _swap(self, i, j):
    self._data[i], self._data[j] = self._data[j], self._data[i]
```

Heap u Pythonu 2

```
def _upheap(self, j):
  parent = self._parent(j)
  if j > 0 and self._data[j] < self._data[parent]:
    self._swap(j, parent)
    self._upheap(parent)

def _downheap(self, j):
  if self._has_left(j):
    left = self._left(j)
    small_child = left
    if self._has_right(j):
      right = self._right(j)
      if self._data[right] < self._data[left]:
        small_child = right
    if self._data[small_child] < self._data[j]:
      self._swap(j, small_child)
      self._downheap(small_child)
```

Heap u Pythonu 3

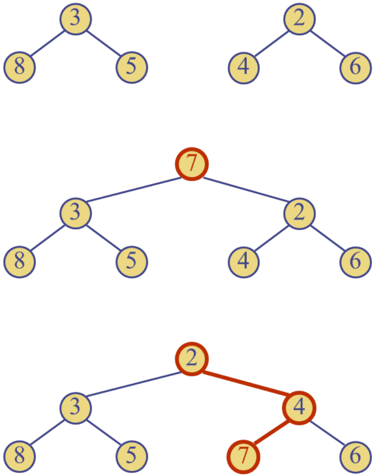
```python
def add(self, key, value):
  self._data.append(PriorityQueueItem(key, value))
  self._upheap(len(self._data)-1)

def min(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  item = self._data[0]
  return (item.key, item.value)

def remove_min(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  self._swap(0, len(self._data)-1)
  item = self._data.pop()
  self._downheap(0)
  return (item.key, item.value)
```

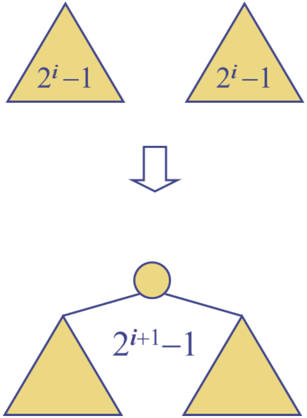
Spajanje dva heapa

- imamo dva heapa i ključ k
- kreiramo novi heap sa korenom k i dva heapa kao podstabla
- pokrenemo **downheap** da restauriramo redosled

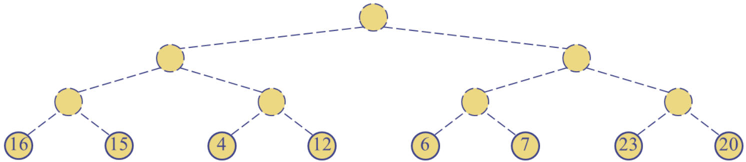


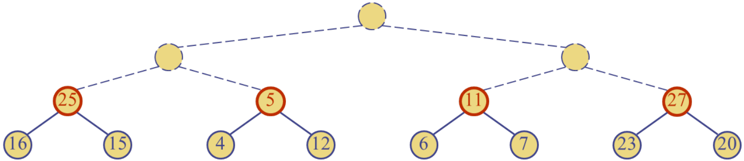
Konstrukcija heapa od dole (bottom-up)

- možemo da napravimo heap sa n ključeva pomoću bottom-up spajanja u $O(\log n)$ koraka
- u i-tom koraku, par heapova sa $2^i - 1$ ključeva se spajaju u heap sa $2^{i+1} - 1$ ključeva

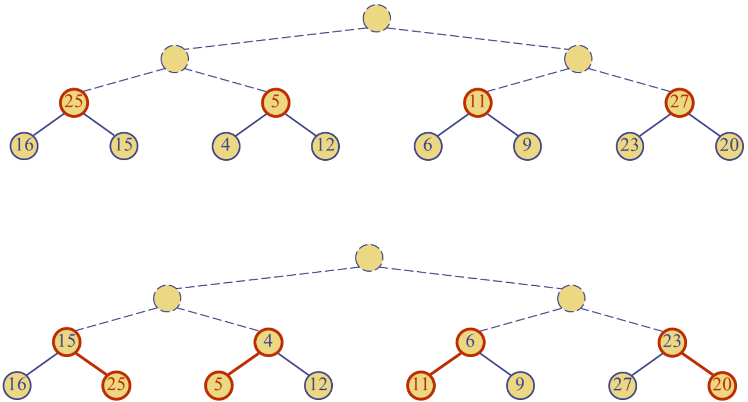


Bottom-up primer $_1$

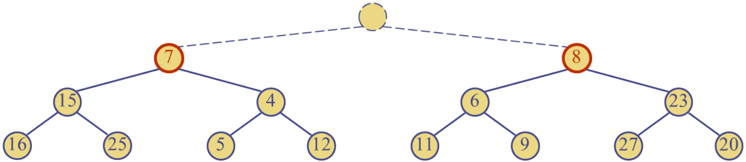


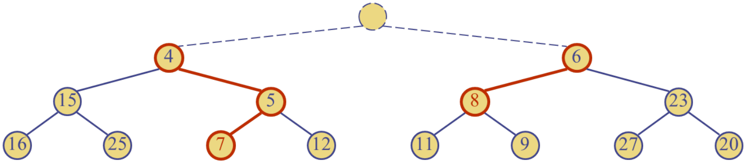


Bottom-up primer $_2$

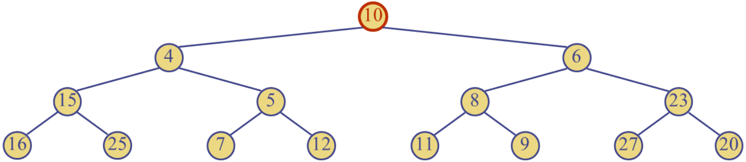


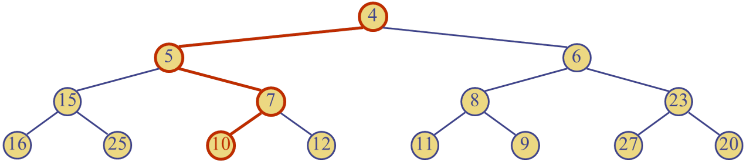
Bottom-up primer $_3$





Bottom-up primer $_4$





Analiza konstrukcije heapa $_1$

- listovi ne narušavaju osobinu heap-a, ne moraju se proveravati: preskočimo kontrolu polovine svih čvorova u stablu
- čvor na nivou h će morati da se pomera najviše h puta - $O(h)$
- ukupan broj čvorova na nivou h je $\lceil \frac{n}{2^{h+1}} \rceil$
- ukupan broj čvorova za sve nivoe je $\sum_{h=0}^{\lfloor \log n \rfloor}$
- n je konstanta – može ispred sume, h može u razlomak, O može ispred:

$$O(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}})$$

- $O(2^{h+1})$ je $O(2^h \cdot 2)$ je $O(2^h)$

Analiza konstrukcije heapa $_2$

- kada $n \to \infty$ suma $\sum_h^\infty \frac{h}{2^h}$ teži 2

$$O(n \cdot \sum_h^\infty \frac{h}{2^h}) = O(n \cdot 2) = O(n)$$

- složenost algoritma je $O(n)$!!!

Adaptivni red sa prioritetom

- **primer**: sistem za kupoprodaju akcija koristi dva reda sa prioritetom, jedan za prodaju i drugi za kupovinu sa elementima (p, s)
    - ključ p je cena
    - vrednost s je broj akcija
    - nalog za kupovinu (p, s) se izvršava kada se pojavi nalog za prodaju (p', s') sa cenom $p' \leq p$ (postupak je završen ako $s' \geq s$)
    - nalog za prodaju (p, s) se izvršava kada se pojavi nalog za kupovinu (p', s') sa cenom $p' \leq p$ (postupak je završen ako $s' \geq s$)
- šta ako neko hoće da otkaže nalog pre nego što se izvrši?
- šta ako neko hoće da izmeni cenu ili broj akcija?

Adaptivni red sa prioritetom: operacije

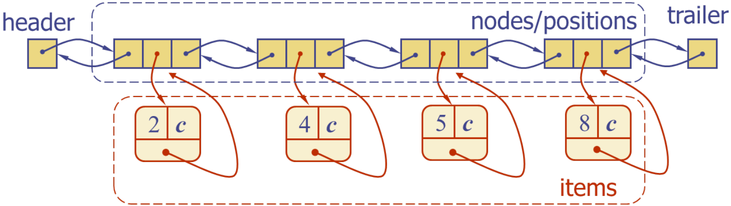
- remove(loc): ukloni i vrati element e iz reda za lokator loc
- update(loc, k, v): zameni ključ/vrednost par (k, v) za lokator loc

Lokatori

- element sa lokatorom identifikuje i prati poziciju (k, v) unutar strukture podataka
- primeri:
  - broj kaputa u garderobi
  - broj rezervacije
- osnovna ideja:
  - pošto elemente kreira i vraća sama struktura podataka, oni mogu biti takvi da pamte svoju lokaciju, što pojednostavljuje kasnije ažuriranje

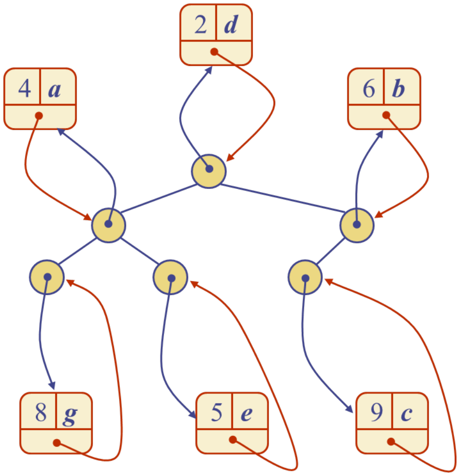
Lokatori i liste

- element liste čuva
  - ključ
  - vrednost
  - poziciju
- reference se ažuriraju u **swap** operaciji



Lokatori i heap

- element heapa čuva
  - ključ
  - vrednost
  - poziciju
- reference se ažuriraju u **swap** operaciji



Performanse

- dobici u brzini usled korišćenja lokatora su označeni crveno

metoda	**nesortirana lista**	**sortirana lista**	**heap**
`len`, `is_empty`	$O(1)$	$O(1)$	$O(1)$
`add`	$O(1)$	$O(n)$	$O(\log n)$
`min`	$O(n)$	$O(1)$	$O(1)$
`remove_min`	$O(n)$	$O(1)$	$O(\log n)$
`remove`	$O(1)$	$O(1)$	$O(\log n)$
`update`	$O(1)$	$O(n)$	$O(\log n)$

Red sa prioritetom i sortiranje

- možemo upotrebiti red sa prioritetom za sortiranje niza elemenata
  - dodamo elemente jedan po jedan putem add operacije
  - uklonimo elemente jedan po jedan putem remove_min operacije
- vreme izvršavanja zavisi od načina implementacije

PQ_sort(S, C)

Input: sekvenca S, komparator C
Output: rastuće sortirana S u skladu sa C

$P \leftarrow$ RSP sa komparatorom C
while $\neg S$.is_empty() **do**
  $e \leftarrow S$.remove_first()
  P.add($e, \emptyset$)
while $\neg P$.is_empty() **do**
  $e \leftarrow P$.remove_min().key()
  S.add_last(e)

Selection sort

- selection sort je varijanta PQ-sorta gde je RSP implementiran pomoću **nesortirane** liste
- vreme izvršavanja selection sorta:
  - dodavanje n elemenata u RSP traje $O(n)$
  - uklanjanje n elemenata u sortiranom redosledu traje

$$1 + 2 + ... + n$$

- selection sort radi u $O(n^2)$ vremenu

Selection sort: primer

	sekvenca S	**red** P
ulaz:	$(7, 4, 8, 2, 5, 3, 9)$	$()$
faza 1		
(a)	$(4, 8, 2, 5, 3, 9)$	(7)
(b)	$(8, 2, 5, 3, 9)$	$(7, 4)$
...	...	...
(g)	$()$	$(7, 4, 8, 2, 5, 3, 9)$
faza 2		
(a)	(2)	$(7, 4, 8, 5, 3, 9)$
(b)	$(2, 3)$	$(7, 4, 8, 5, 9)$
(c)	$(2, 3, 4)$	$(7, 8, 5, 9)$
(d)	$(2, 3, 4, 5)$	$(7, 8, 9)$
(e)	$(2, 3, 4, 5, 7)$	$(8, 9)$
(f)	$(2, 3, 4, 5, 7, 8)$	(9)
(g)	$(2, 3, 4, 5, 7, 8, 9)$	$()$

Insertion sort

- insertion sort je varijanta PQ-sorta gde je RSP implementiran pomoću **sortirane** liste
- vreme izvršavanja insertion sorta:
  - dodavanje n elemenata u RSP traje

$$1 + 2 + \ldots + n$$

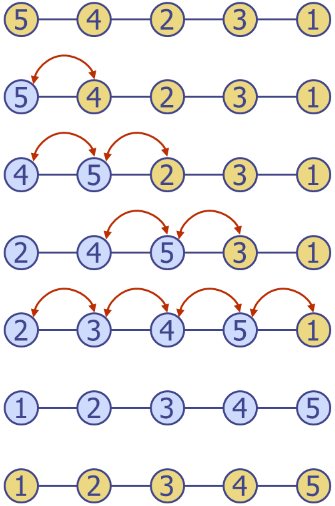
  - uklanjanje n elemenata traje $O(n)$
- insertion sort radi u $O(n^2)$ vremenu

Insertion sort: primer

	sekvenca S	**red** P
ulaz:	$(7,4,8,2,5,3,9)$	$()$
faza 1		
(a)	$(4,8,2,5,3,9)$	(7)
(b)	$(8,2,5,3,9)$	$(4,7)$
(c)	$(2,5,3,9)$	$(4,7,8)$
(d)	$(5,3,9)$	$(2,4,7,8)$
(e)	$(3,9)$	$(2,4,5,7,8)$
(f)	(9)	$(2,3,4,5,7,8)$
(g)	$()$	$(2,3,4,5,7,8,9)$
faza 2		
(a)	(2)	$(3,4,5,7,8,9)$
(b)	$(2,3)$	$(4,5,7,8,9)$
...	...	...
(g)	$(2,3,4,5,7,8,9)$	$()$

Sortiranje unutar iste strukture podataka (in-place)

- umesto korišćenja 2 strukture možemo implementirati selection i insertion sort u okviru jedne strukture
- deo ulaznog niza će poslužiti kao RSP
- za insertion sort
  - držimo sortiran početak niza
  - elemente menjamo pomoću swap operacije



Heap sort

- posmatramo RSP sa n elemenata, implementiran pomoću heapa
  - potreban prostor je $O(n)$
  - add i remove_min traju $O(\log n)$
  - len, is_empty, min traju $O(1)$
- ovakav RSP možemo koristiti za sortiranje n elemenata za $O(n \log n)$ vreme
- rezultujući algoritam se zove heap sort
- znatno brži od kvadratnih algoritama kao što su selection i insertion sort

$$O(n \log n) < O(n^2)$$