Heapovi

Miloš Stanojević

Matematička gimnazija, Nedelja informatike

31. mart 2015.

Često nas, u sklopu kako takmičarskih tako i ostalih problema u računarstvu, interesuje da efikasno održavamo neki skup podataka sa sledećim svojstvima:

- Brz pristup "najvažnijem" elementu/elementu najvećeg prioriteta;
- Brzo dodavanje novih elemenata;
- Brzo uklanjanje "najvažnijeg" elementa;

NB Jedna takva struktura je **priority_queue** (C++)...

Motivacija - šta je prioritetni red?

Prioritetni red ili priority queue (u daljem tekstu **pq**), je struktura koja implementira sledeće metode:

- **Item* first()** - vraća element sa najmanjim ključem;
- **Item* extractMin()** - vraća element sa najmanjim ključem i uklanja ga iz prioritetnog reda;
- **void insert(Item* x)** - dodaje novi element u prioritetni red;
- **void decreaseKey(Item* x, Key k)** - smanjuje vrednost ključa elementa x na novu vrednost k, time mu povećavši značaj;
- **void delete(Item* x)** - uklanja odabrani element x iz prioritetnog reda;
- **pq* merge(pq* q1, pq* q2)** - spaja dva prioritetna reda u jedan i pritom ih oba uništava.

NB Struktura **Item** sadrži ključ (tipa **Key**) i neki sadržaj.

Problem prodavaca kupusa

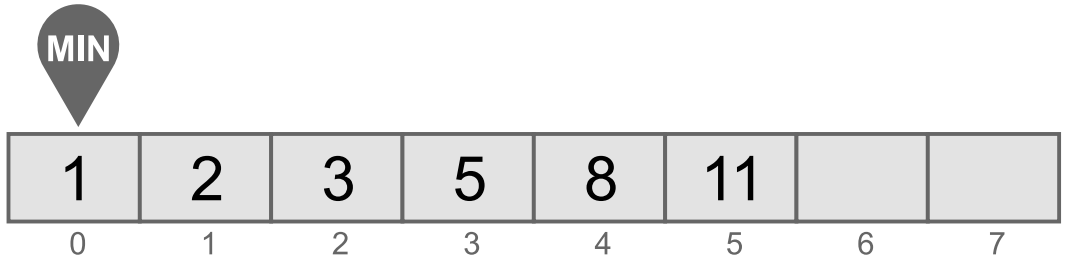
Srpski prodavci kupusa su suočeni sa sledećom situacijom:

- Postoji n velikih pijaca kupusa, na kojima oni prodaju kupus;
- Svaki prodavac se odlučio za cenu po kojoj prodaje svoj kupus;
- Na svakoj pijaci se kupus prodaje odvojeno, i kupci uvek kupuju najjeftiniji kupus, pošto je sav kupus istog kvaliteta;
- Sve što se na pijacama ne proda na domaćem tržištu, na kraju sezone objedinjeno se nudi kupcima iz inostranstva.

Prvo rešenje

Koristimo **niz** kao osnovu našeg prioritetnog reda:

- Niz sve vreme održavamo sortiranim (bubble/insertion);
- Prvi element ima najmanji ključ (odnosno cenu u našem primeru);
- Metoda **first()** ima konstantnu složenost, a sve ostale $O(n)$.



Pregled složenosti rešenja

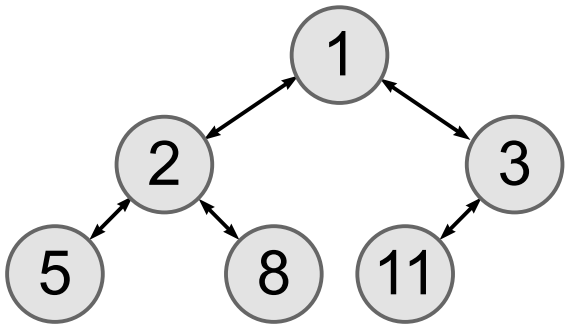
Operacija	Složenosti (niz)
Pravljenje praznog pq-a	$O(1)$
first()	$O(1)$
insert()	$O(n)$
extractMin()	$O(n)$
decreaseKey()	$O(n)$
delete()	$O(n)$
merge()	$O(n)$

Binarni heap - Motivacija

- Implementacija prioritetnog reda koja koristi niz laka je za implementiranje;
- Složenosti su nažalost linearne, možemo i bolje od toga;
- Pokušajmo da to popravimo drugačijom strukturom podataka, sa sledećim svojstvima...

Bitna svojstva

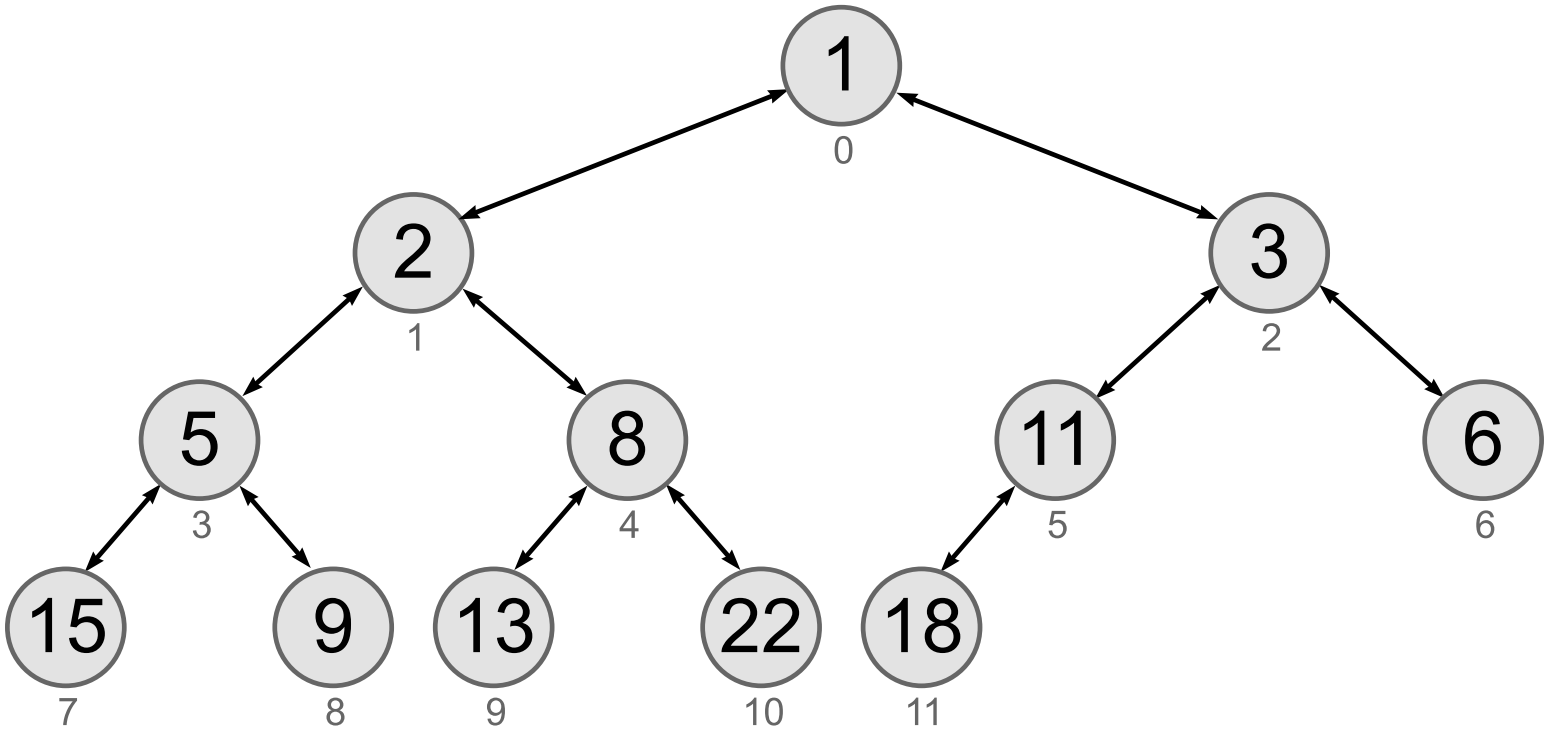
- **Heap svojstvo:** Ključ svakog roditelja je veći ili jednak od ključeva njegove dece;
- **Svojstvo kompletnosti:** Sve dubine drveta, osim potencijalno najdublje, potpuno su popunjene;
- **Binarno drvo** koje zadovoljava heap svojstvo i svojstvo kompletnosti zvaćemo **binarni heap**.



Osobine binarnog heapa

- Koreni element je najmanji (posledica heap svojstva);
- Dubina (skoro potpunog) drveta je najviše $\lceil \log(n) \rceil$ ukoliko je n ukupan broj elemenata u drvetu;
- Ubacivanje i uklanjanje elementa se dakle može obaviti u $O(\log(n))$, dokle god je drvo skoro potpuno;
- Jedini problem je spajanje dva binarna heapa, što najbolje možemo odraditi u $O(n)$ (kako?).

Primer binarnog heapa



Izgradnja heapa u $O(n)$

Prvo moramo rešiti podproblem: Kako izgraditi jedan heap od datih n elemenata u linearnoj složenosti?

- Prvi pokušaj: samo ih sve poubacujemo unutra jedan po jedan, koristeći **insert()**;
- Ali složenost je onda $O(n \log(n)) > O(n)$, moramo bolje;
- Pokušajmo umesto toga da heap gradimo odozdo...

Izgradnja heapa u $O(n)$, cont'd

- Prvo prepišimo sve elemente u niz ($O(n)$);
- Posmatrajmo ovaj niz kao binarno drvo, gde ukoliko je roditelj na poziciji i, deca su na $2 \cdot i + 1$ i $2 \cdot i + 2$;
- Krenimo sa dna i "korigujmo" binarno drvo tako da zadovolji heap svojstvo;
- Zašto je ovo rešenje u $O(n)$?

Šta je uopšte složenost?

Kako procenjujemo koliko je nekom algoritmu potrebno da se izvrši, ako ne znamo kakav mu je unos? Uvedimo prvo par uprošćenja:

- Interesuje nas samo najgore (najduže) moguće vreme potrebno da se algoritam izvrši;
- Ne merimo apsolutno vreme, već samo *stopu rasta* i ignorišemo konstante;
- Bilo koji konačan broj izuzetaka od naše procene je nebitan, dokle god je ona tačna za sva "dovoljno velika" n;
- Ne ograničavamo se samo na razumne vrednosti za n, ili primenjujemo bilo kakve druge provere smislenosti.

Sa ovim ograničenjima na umu uvedimo sada, malo formalnije, veliko-O notaciju…

Ukratko o veliko-O notaciji

Za funkciju $f(n)$ kažemo da je $O(g(n))$ ukoliko postoje konstante k i N, za koje važi:

$$0 \leq f(n) \leq k \cdot g(n) \text{ kada } n > N$$

Tada funkcija $g(n)$ predstavlja **gornju granicu** funkcije $f(n)$, za dovoljno velike vrednosti n i do na konstantan faktor. Ili manje formalno:

$$f(n) \text{ raste } \textit{najviše} \text{ kao } g(n)$$

I, ponovo neformalno, pišemo:

$$f(n) = O(g(n))$$

NB Ovo je zloupotreba matematičke notacije! $O(g(n))$ predstavlja čitavu klasu matematičkih funkcija, pa je pravilnije napisti:

$$f(n) \in O(g(n))$$

Analiza složenosti izgradnje heapa

Primenimo sada ono što smo naučili o veliko-O notaciji na izgradnju heapa:

- Uzmimo visinu drveta kao $h = \log(n)$;
- Onda svaka "korekcija" na dubini j ima cenu od najviše $k(h - j)$, gde je k neka konstanta;
- Dakle, cena da "korigujemo" jedan ceo nivo (koji sadrži 2^j elemenata) je $k(h - j) \cdot 2^j$

Analiza složenosti izgradnje heapa, cont'd

Onda je cena izgradnje celog heapa:

$$C(h) = \sum_{j=0}^{h} k\,(h-j)\cdot 2^j \tag{1}$$

$$= k\frac{2^h}{2^h}\sum_{j=0}^{h} 2^j\,(h-j) \tag{2}$$

$$= k2^h\sum_{j=0}^{h} 2^{j-h}\,(h-j) \tag{3}$$

$$\ldots l = h - j \ldots \tag{4}$$

$$= k2^h\sum_{l=0}^{h} 2^{-l}\,l \tag{5}$$

Analiza složenosti izgradnje heapa, cont'd

$$C(h) = k2^h \sum_{l=0}^{h} l \left(\frac{1}{2}\right)^l \tag{6}$$

S obzirom na to da ova suma konvergira ka $\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$, to znači da $C(h)$ raste najviše kao $O(2^h)$, a pošto je $h = \log(n)$, konačno dobijamo:

$$C(n) = O(n)$$

Spajanje dva binarna heapa

Dakle, dva binarna heapa spajamo u $O(n)$, gde je n ukupan broj elemenata, na sledeći način:

- U $O(n)$ prepišemo elemente oba niza u jedan niz dužine n;
- U $O(n)$ izgradimo od ovog niza novi heap;
- And we are done!

Kratak osvrt na implementaciju

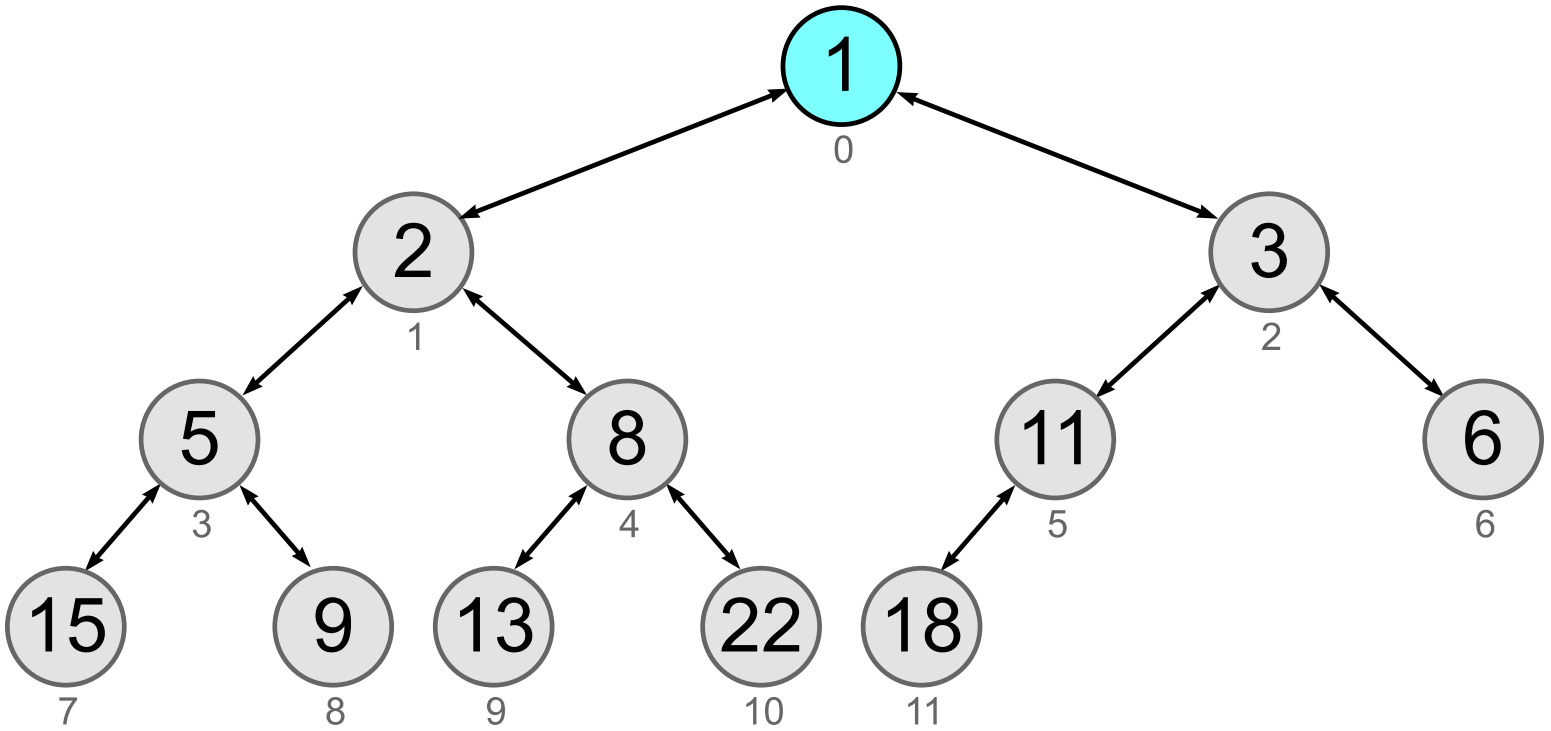
Sada ćemo se nakratko osvrnuti na implementaciju svake metode prioritetnog reda na primeru binarnog heapa (u C++), kako bismo ga bolje razumeli. Binarni heap ćemo definisati kao niz, sa brojem elemenata n, na sledeći način:

```
1 struct Item
2 {
3     int key;
4     char value;
5 };
6
7 Item* heap[1000000];
8 int n = 0;
```

createHeap() i first()

```
1  void createHeap(Item* &heap[], int &n)
2  {
3      n = 0;
4  }
5
6  Item* first(Item* &heap[], int &n)
7  {
8      return heap[0];
9  }
```

first()
korak: 1



extractMin() - Pomoćne metode

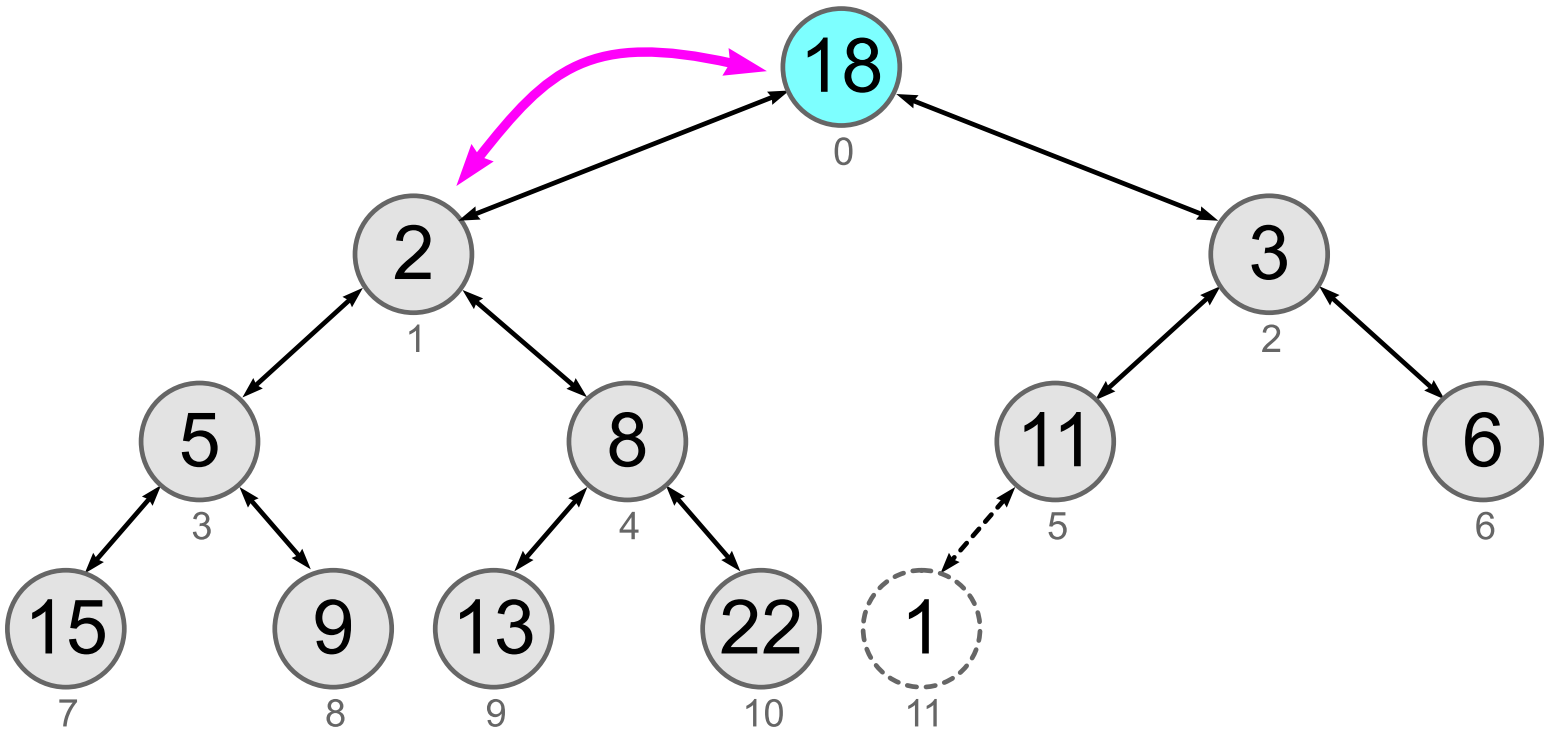
```
1  int choose(int lef, int rig, Item* &heap[], int &n)
2  {
3      if(lef < n)
4      {
5          if(rig < n && heap[rig] -> key < heap[lef] -> key)
6              return rig;
7          return lef;
8      }
9      return -1;
10 }
11
12 void swap(int fir, int sec, Item* &heap[])
13 {
14     Item* pom = heap[fir];
15     heap[fir] = heap[sec];
16     heap[sec] = pom;
17 }
```

extractMin()

```
1  void rebalance(int start_pos, Item* &heap[], int &n)
2  {
3      int choice = choose(2*start_pos + 1, 2*start_pos + 2,
        heap, n);
4      while(choice > 0)
5      {
6          int pos = (choice - 1)/2;
7          if(heap[pos] -> key > heap[choice] -> key)
8              swap(pos, choice, heap);
9          else
10             break;
11         choice = choose(2*choice + 1, 2*choice + 2, heap, n);
12     }
13 }
14 Item* extractMin(Item* &heap[], int &n)
15 {
16     n--;
17     swap(0, n, heap);
18     rebalance(0, heap, n);
19     return heap[n];
20 }
```

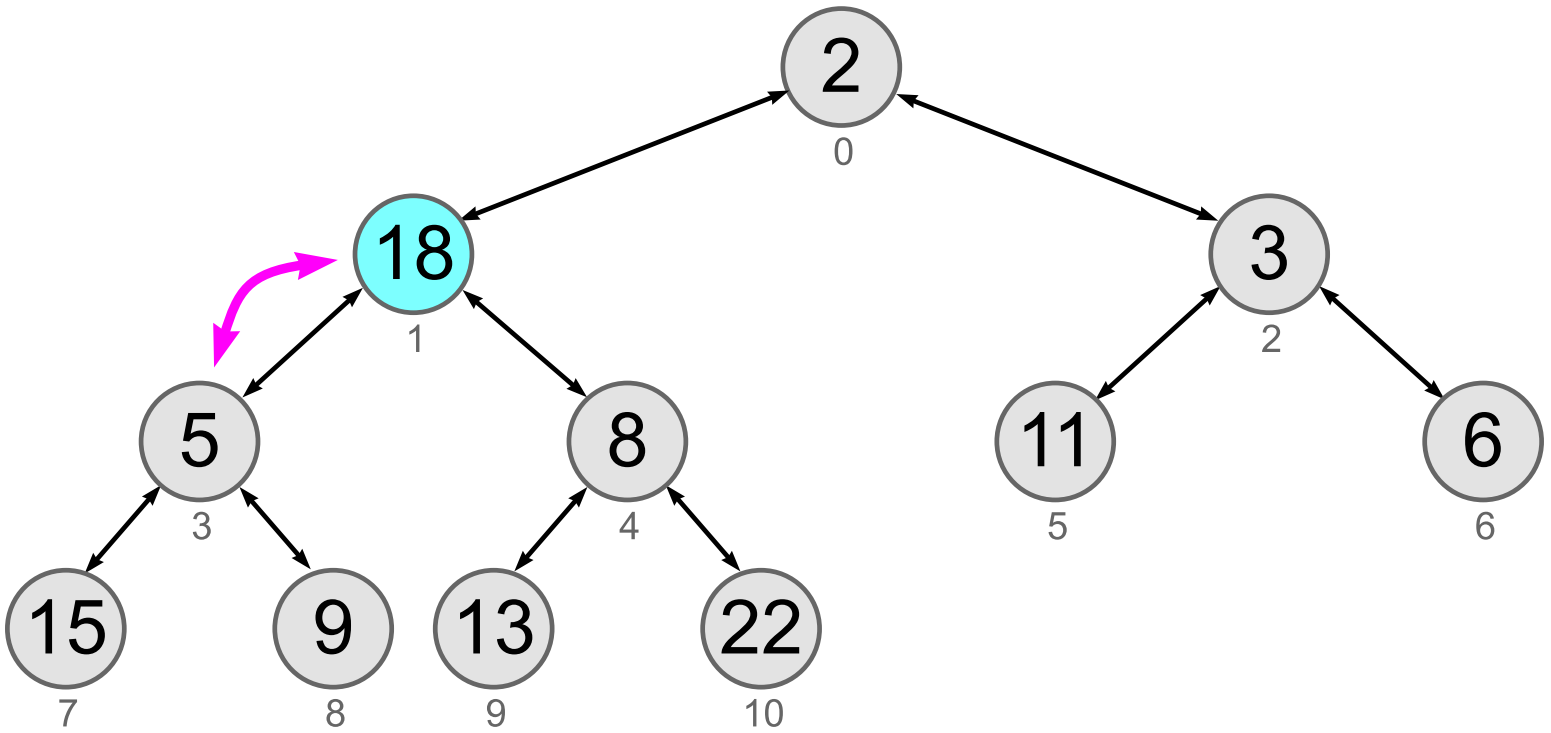
extractMin()

korak: 1



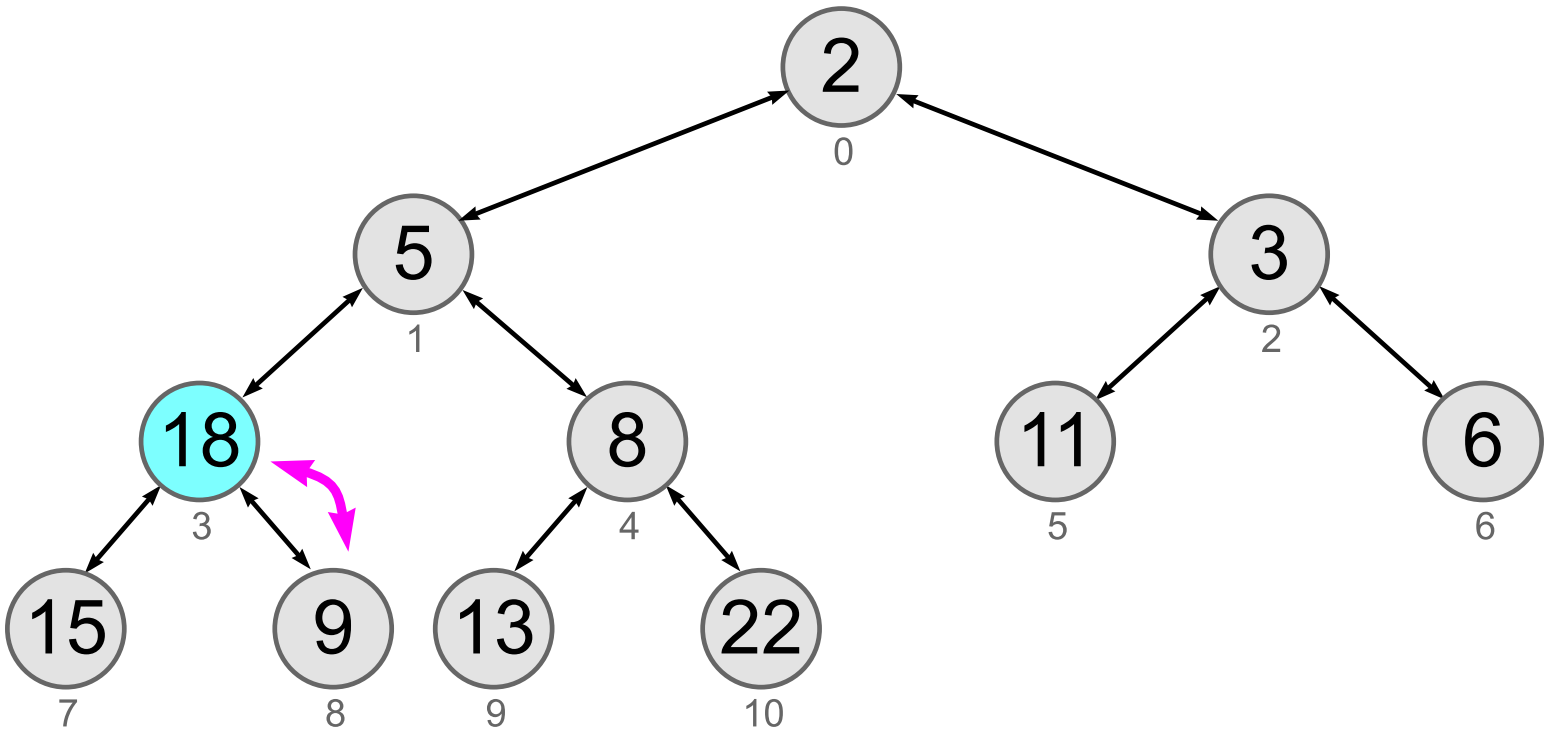
extractMin()

korak: 2



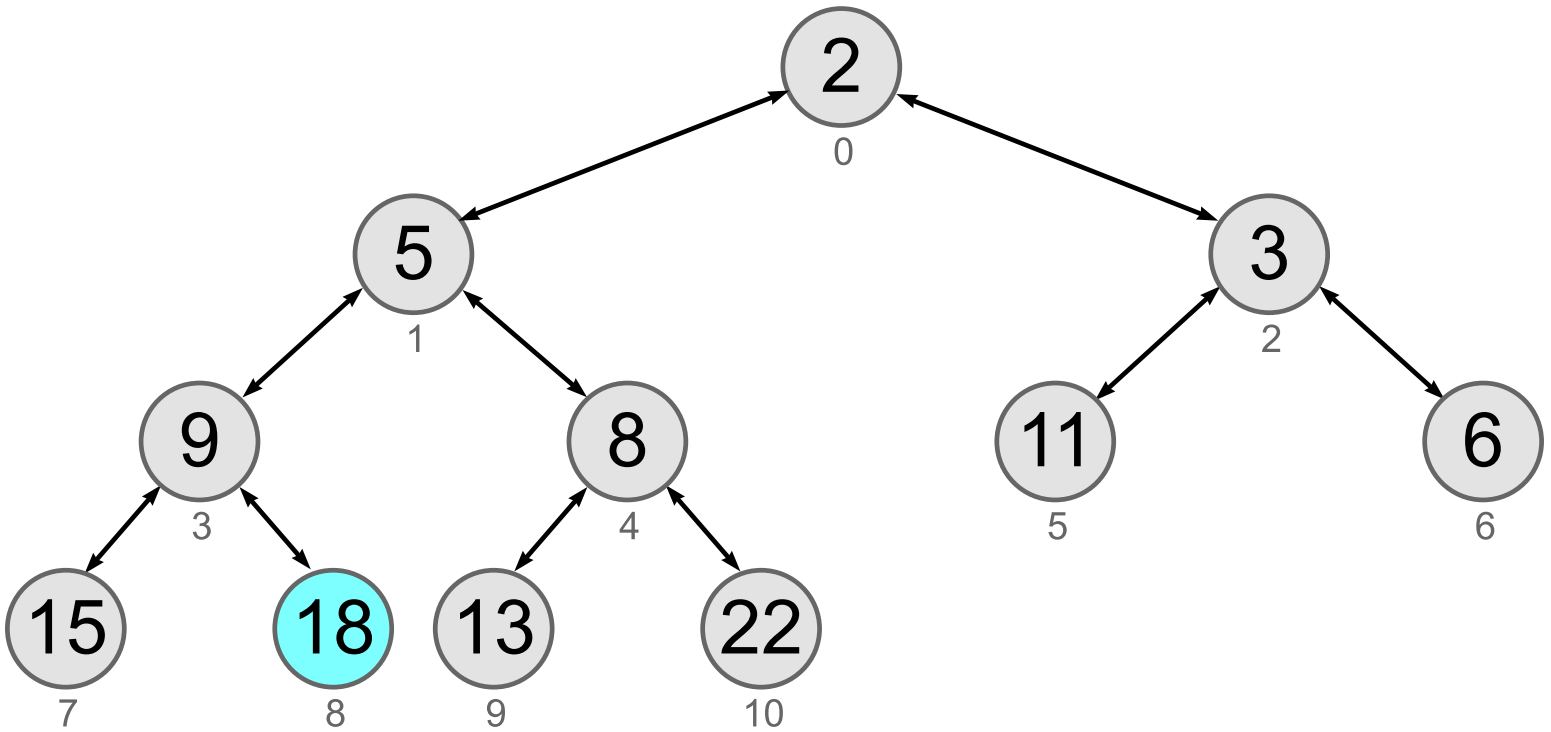
extractMin()

korak: 3



extractMin()

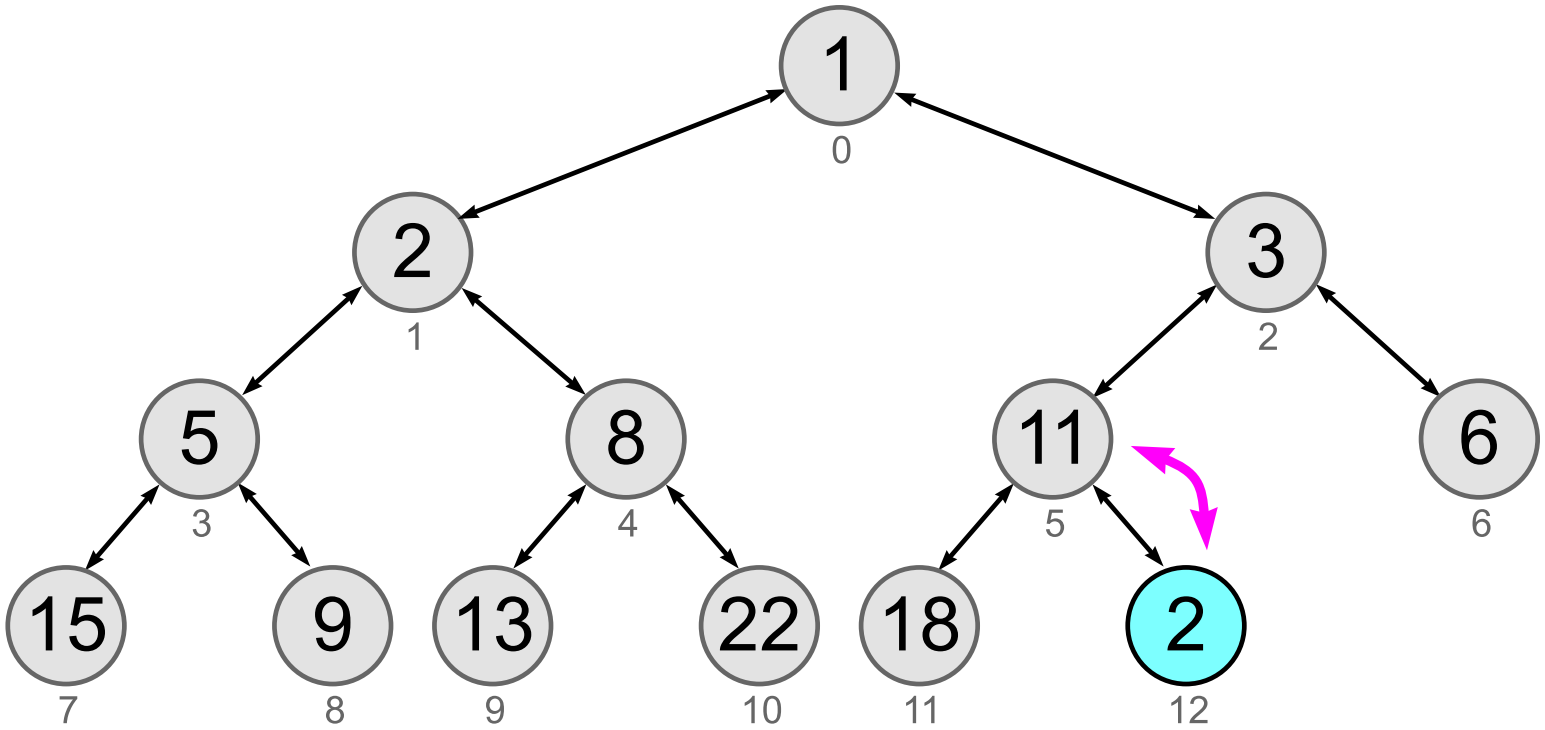
korak: 4



insert()

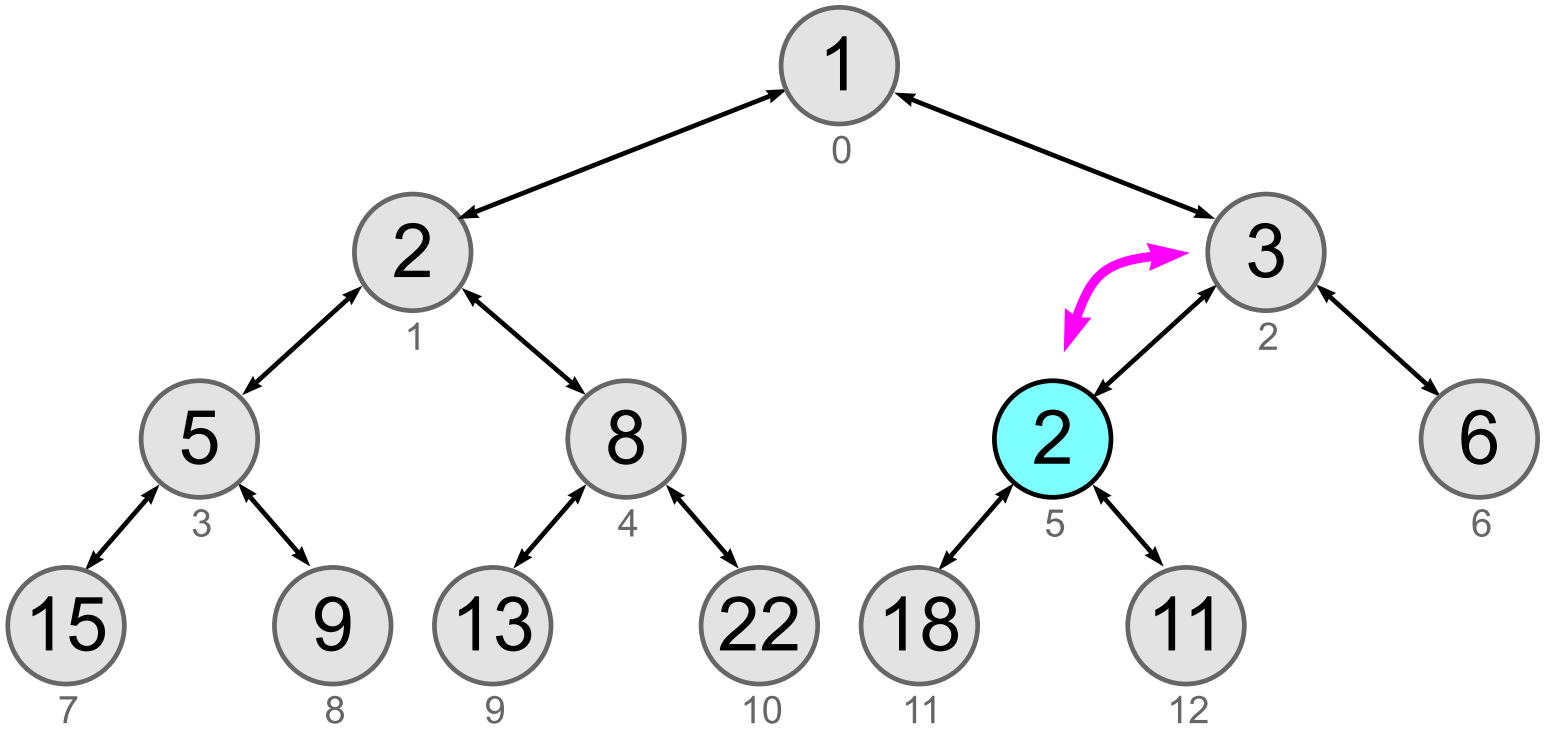
```
1  void insert(Item* x, Item* &heap[], int &n)
2  {
3      heap[n] = x;
4      int pos = n;
5      while((pos - 1)/2 >= 0 && heap[pos] < heap[(pos - 1)/2])
6      {
7          swap(pos, (pos - 1)/2, heap);
8          pos = (pos - 1)/2;
9      }
10     n++;
11 }
```

insert(2)
korak: 1



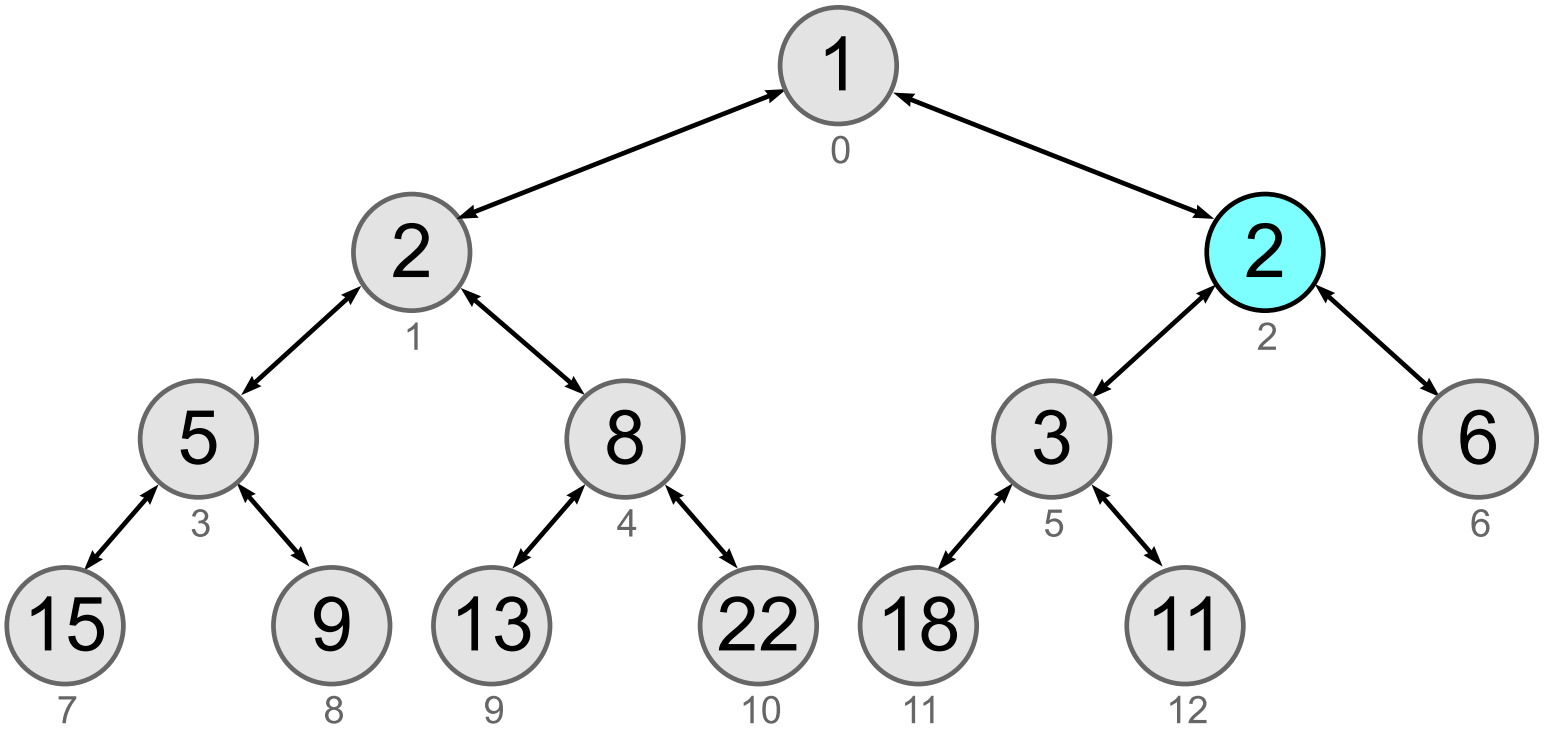
insert(2)

korak: 2



insert(2)

korak: 3

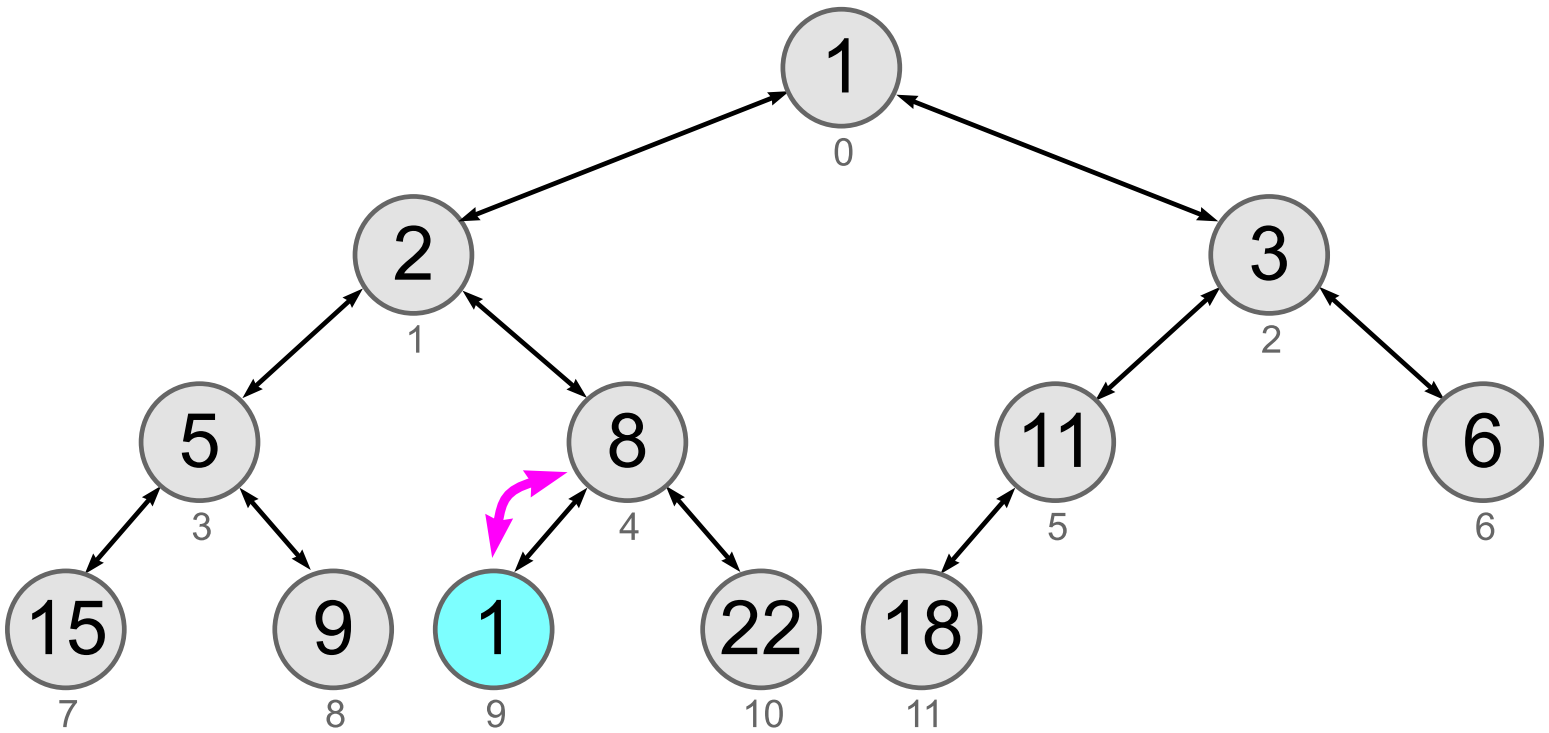


decreaseKey() i delete()

```
1  void decreaseKey(int pos, int newk, Item* &heap[], int &n)
2  {
3      if(newk >= heap[pos] -> key)
4          break;
5      heap[pos] -> key = newk;
6      while((pos - 1)/2 >= 0 && heap[pos] < heap[(pos - 1)/2])
7      {
8          swap(pos, (pos - 1)/2, heap);
9          pos = (pos - 1)/2;
10     }
11 }
12
13 void Delete(int pos, Item* &heap[], int &n)
14 {
15     n--;
16     swap(pos, n, heap);
17     rebalance(pos, heap, n);
18 }
```

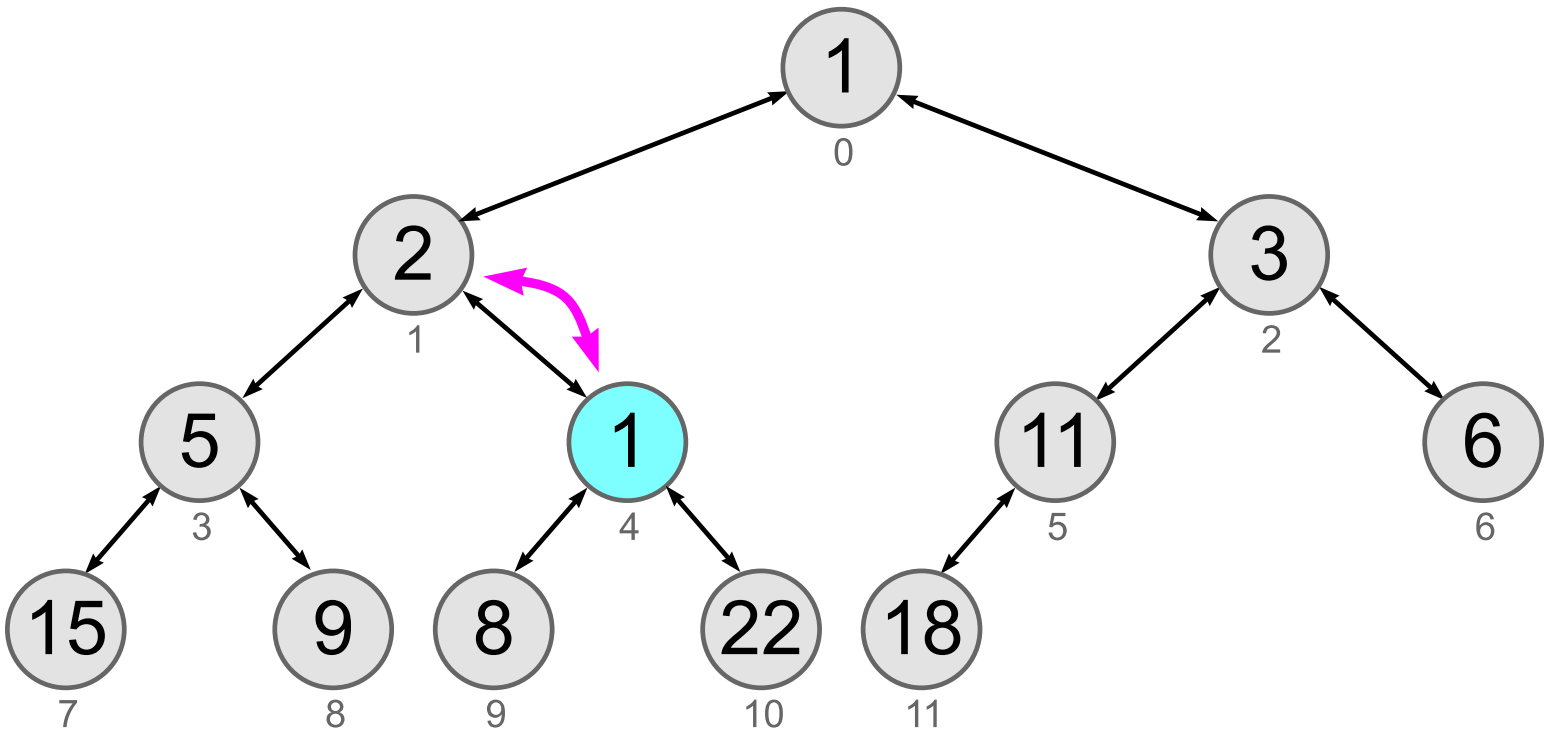
decreaseKey(9, 1)

korak: 1



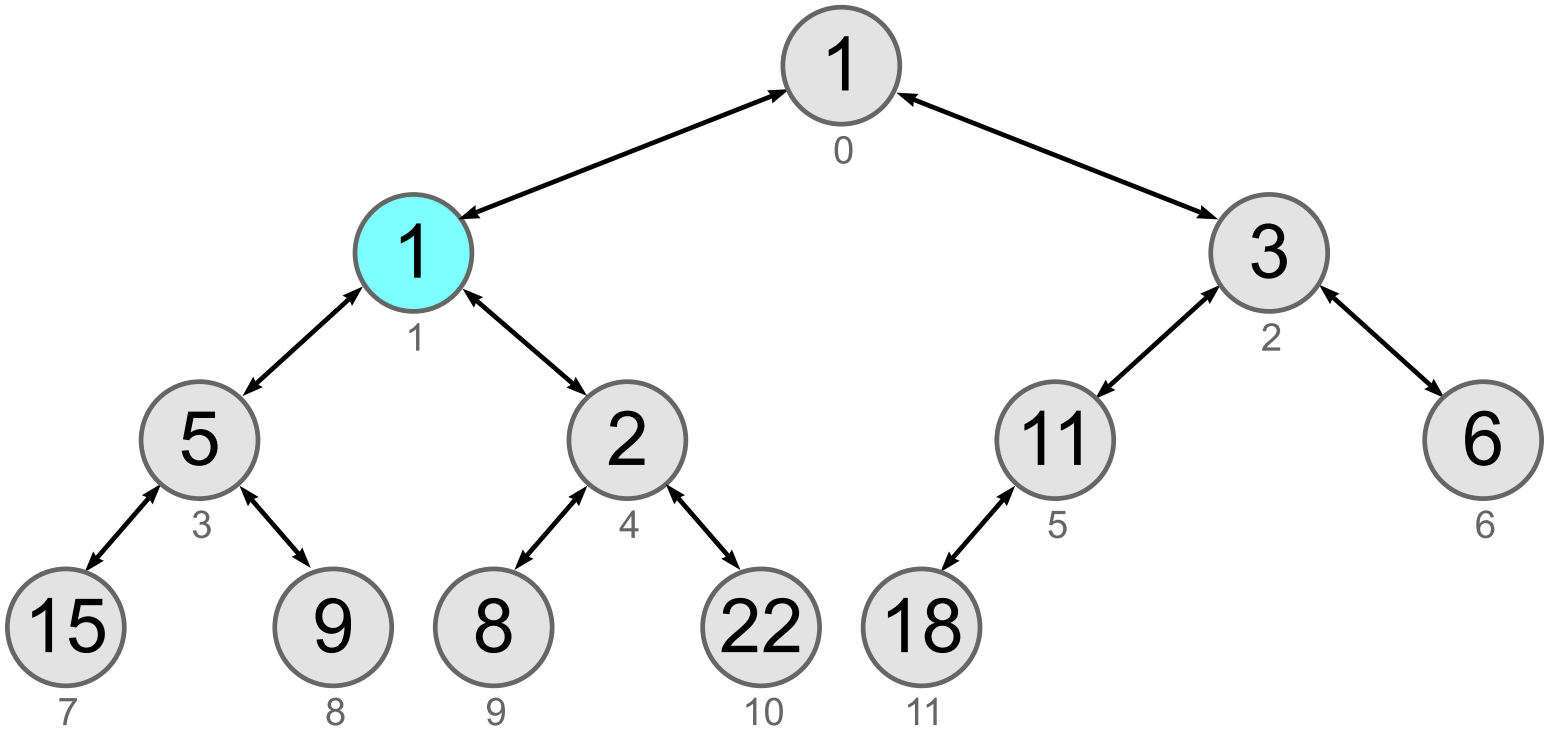
decreaseKey(9, 1)

korak: 2

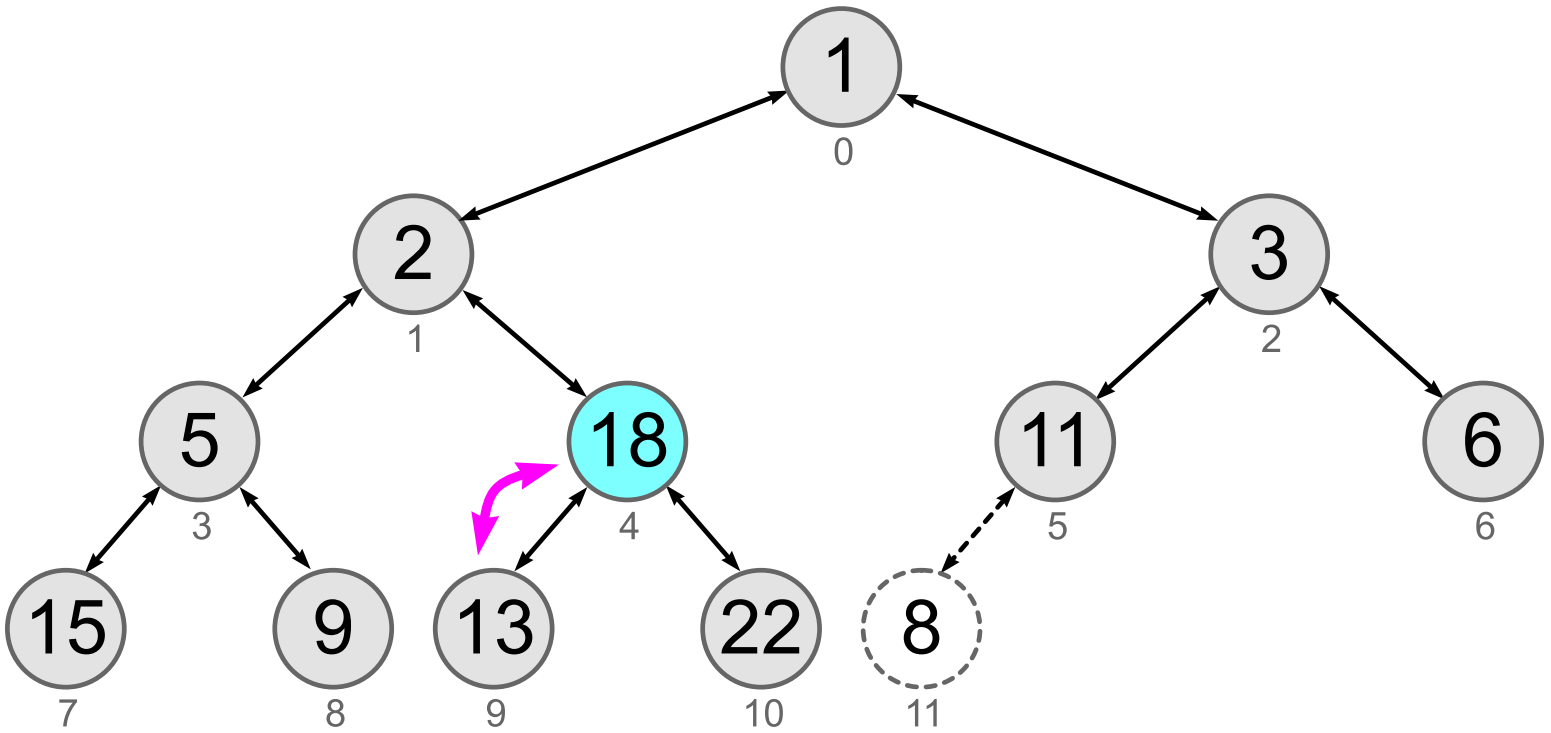


decreaseKey(9, 1)

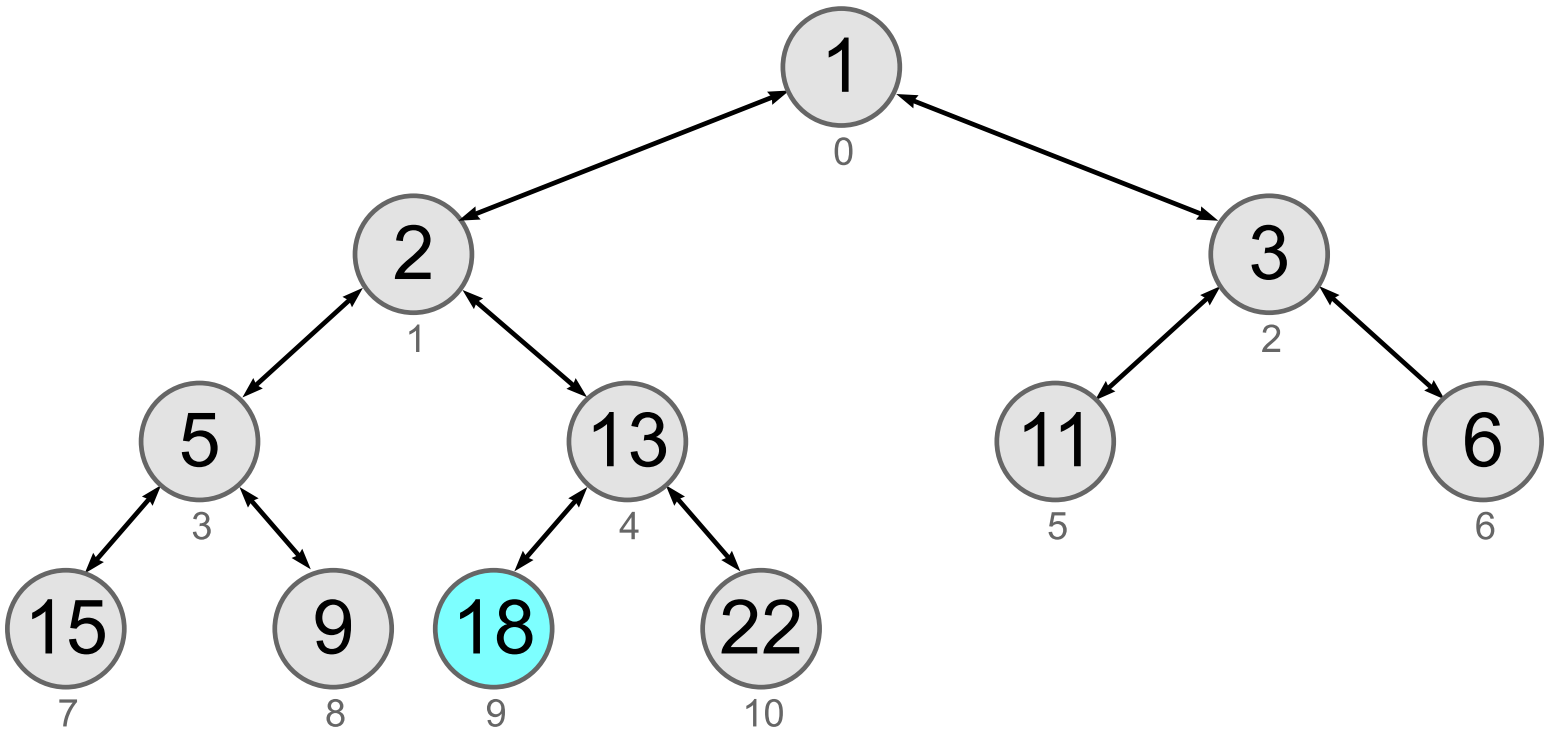
korak: 3



delete(4)
korak: 1



delete(4)
korak: 2

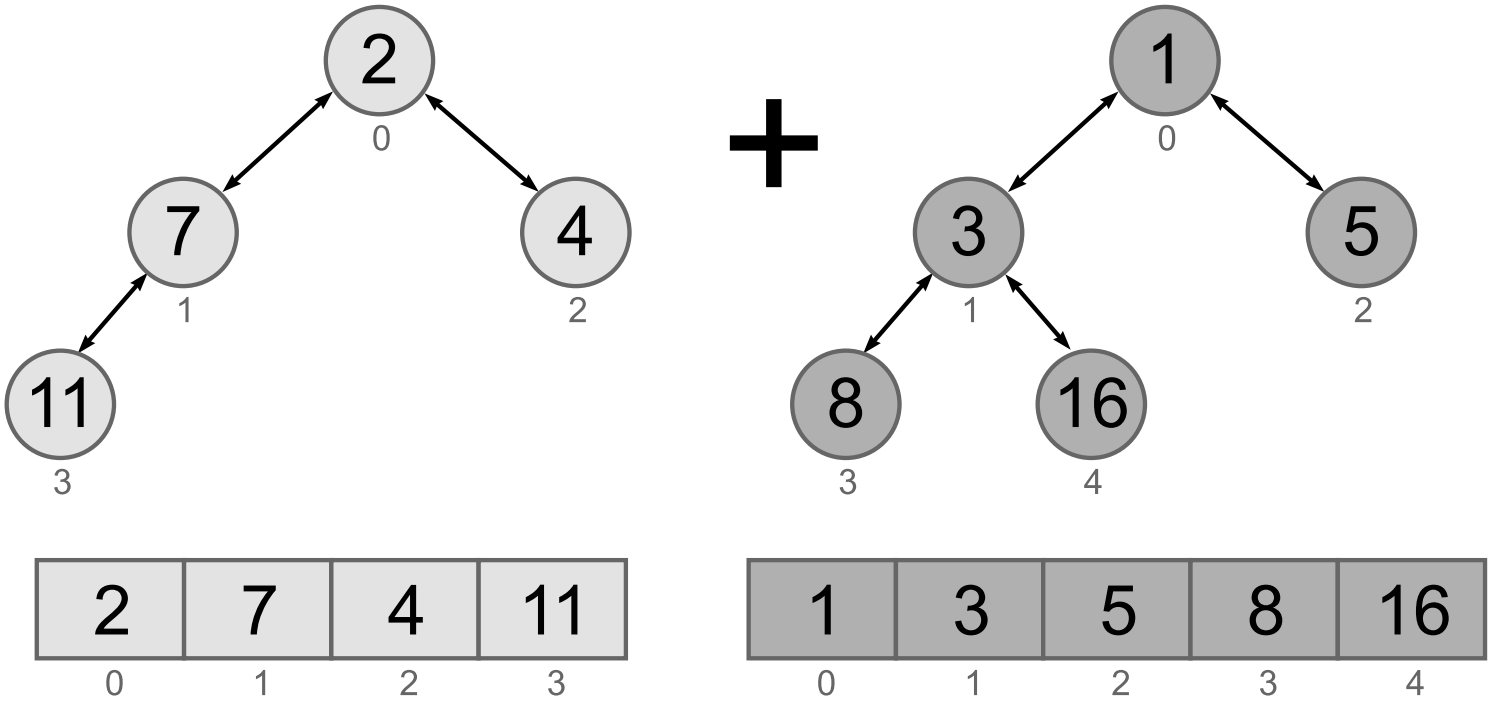


merge()

```
1  void merge(Item* &heap1[], int &n1, Item* &heap2[], int &n2,
       Item* &Heap, int &n)
2  {
3      n = n1 + n2;
4
5      for(int i = 0; i < n1 ; i++)
6          Heap[i] = heap1[i];
7      for(int i = 0; i < n2 ; i++)
8          Heap[n1+i] = heap2[i];
9
10     for(int i = (n-1)/2; i >= 0; i--)
11         rebalance(i, Heap, n);
12 }
```

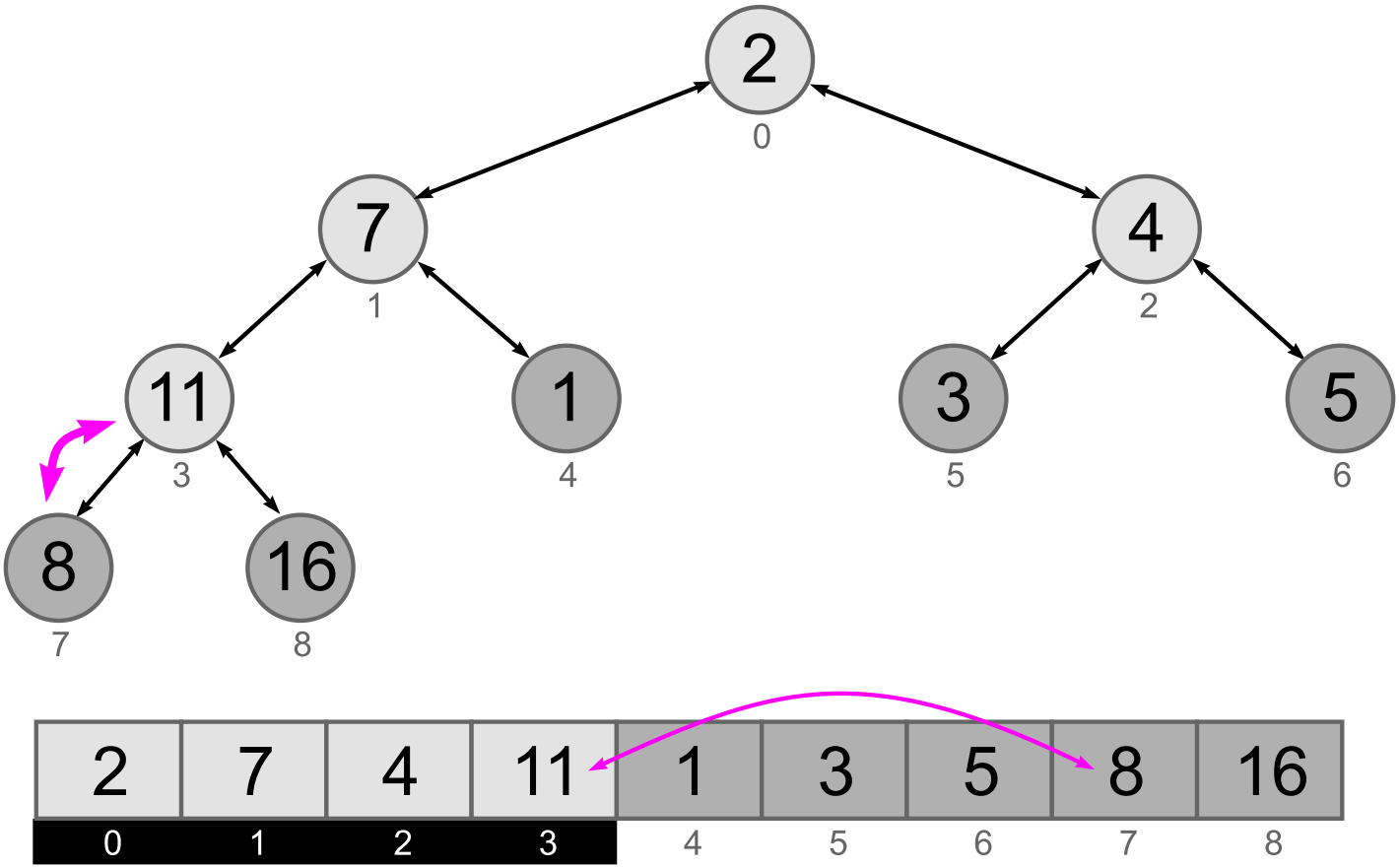
merge(pq1, pq2)

korak: 1



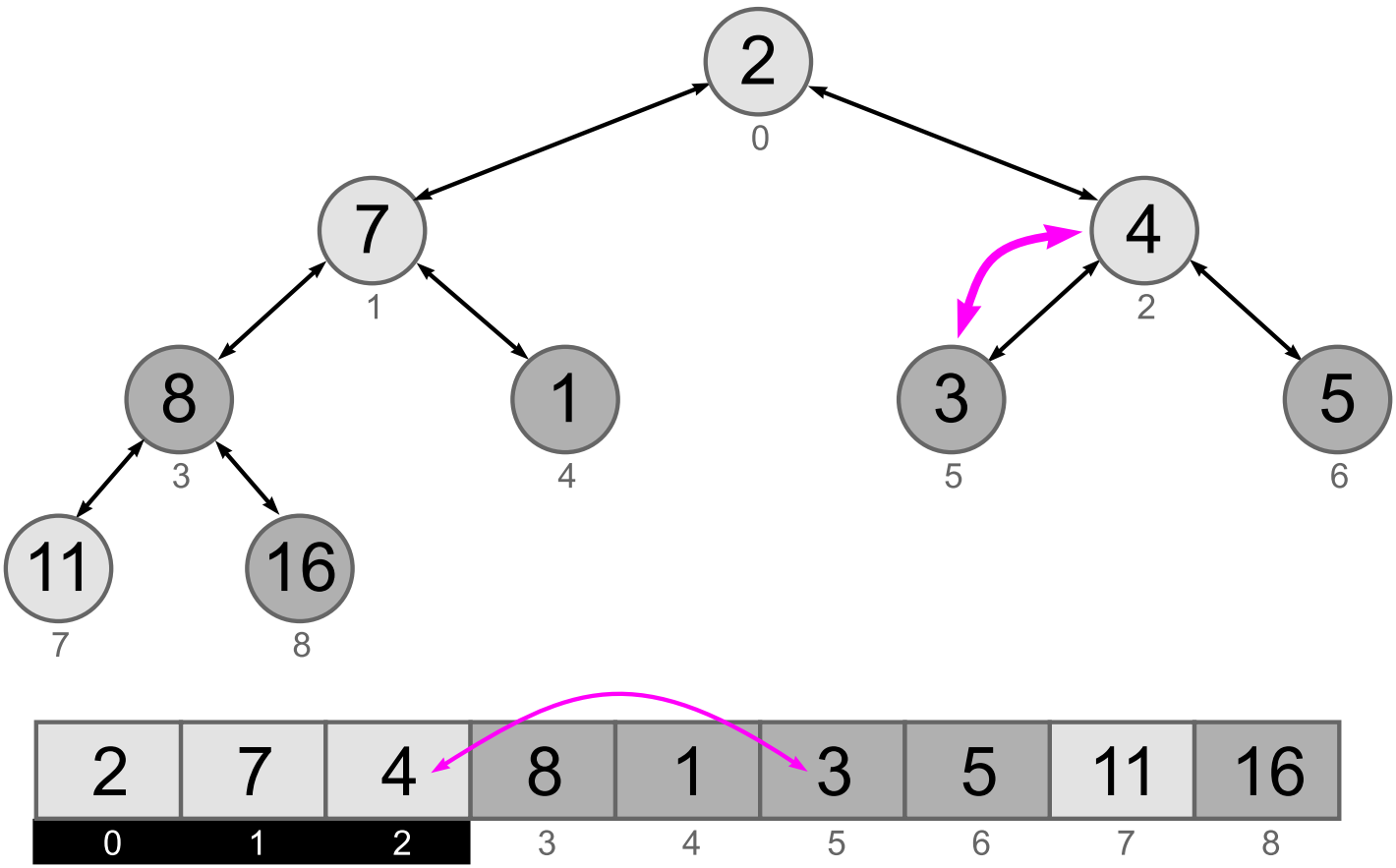
merge(pq1, pq2)

korak: 2



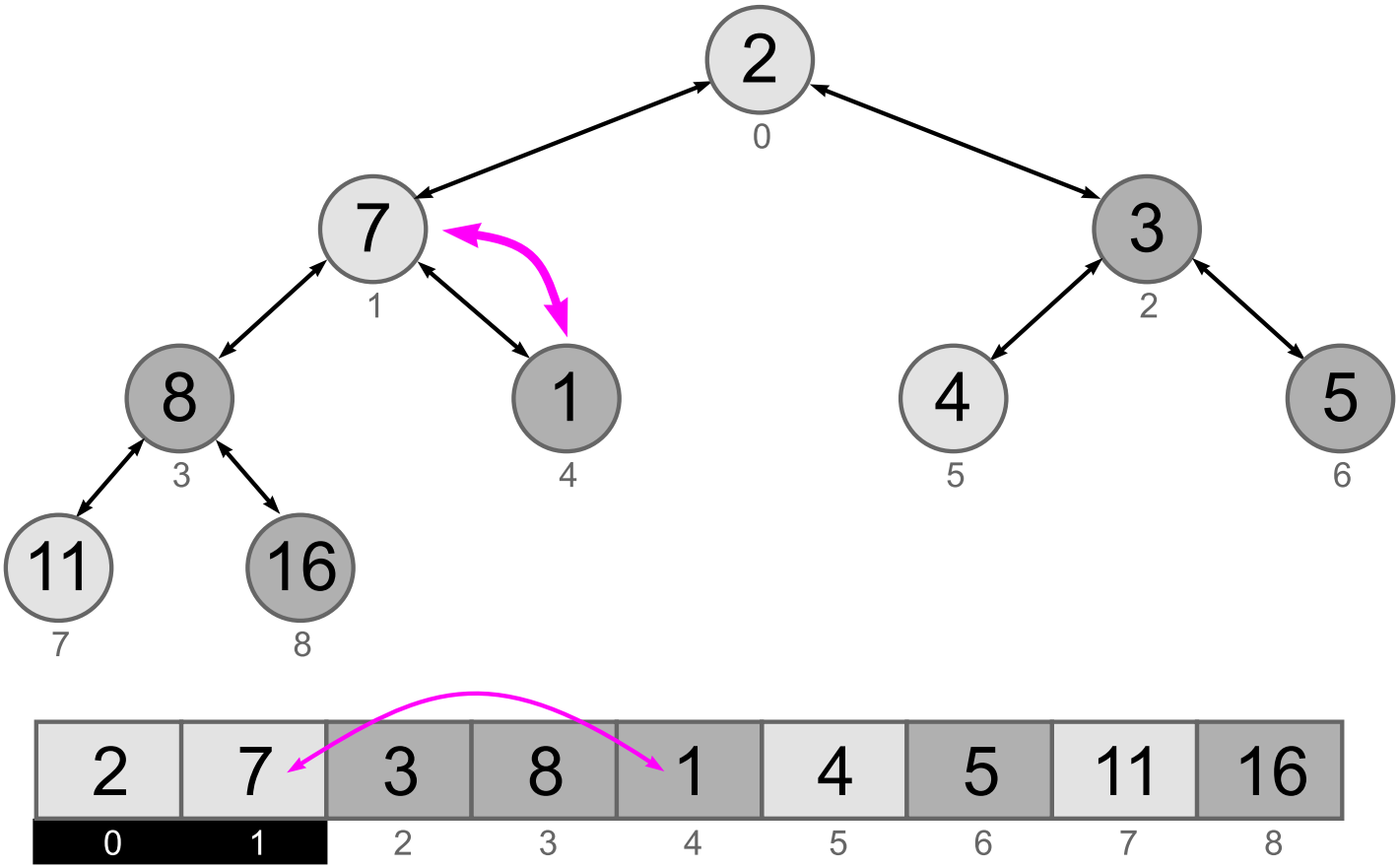
merge(pq1, pq2)

korak: 3



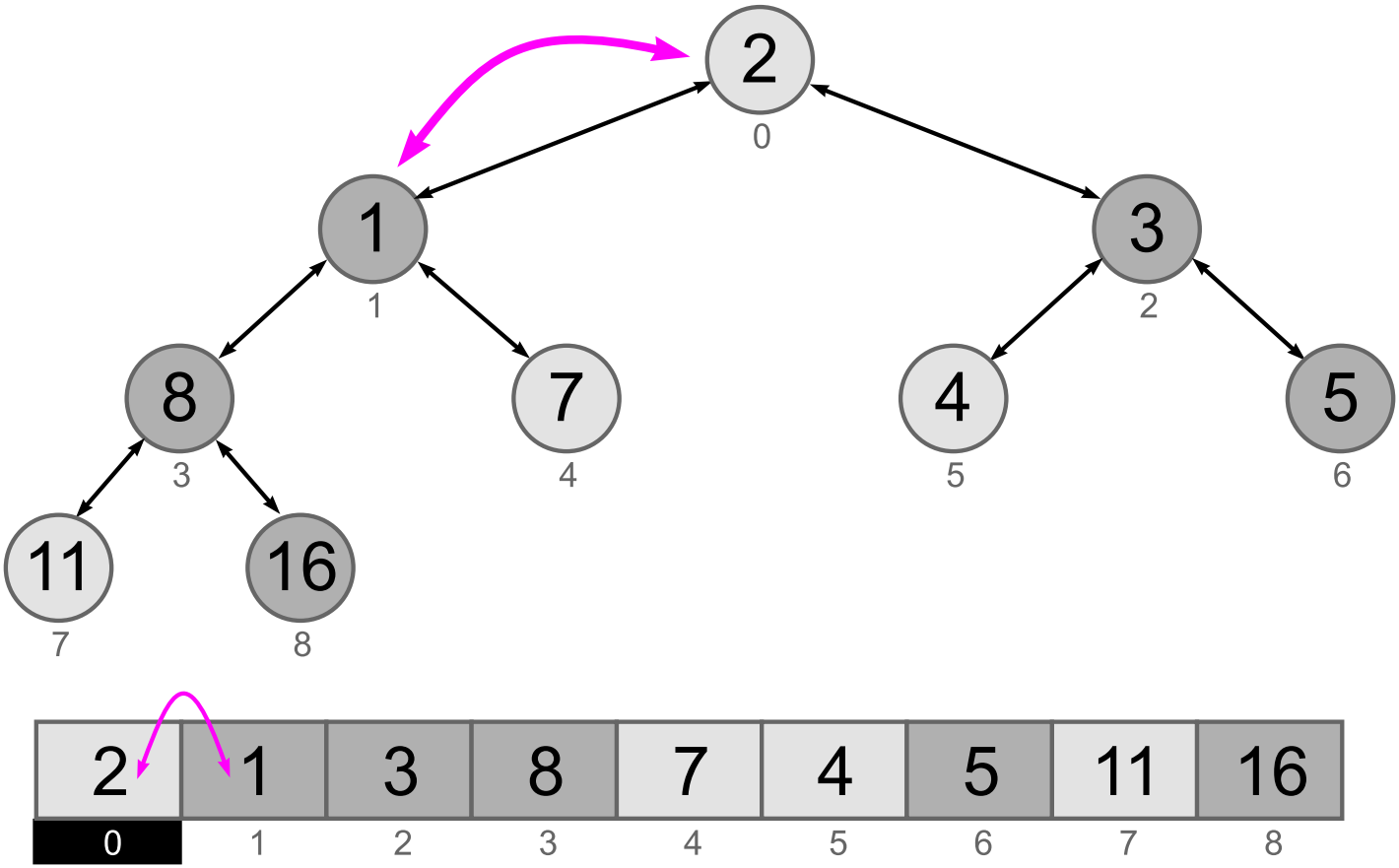
merge(pq1, pq2)

korak: 4



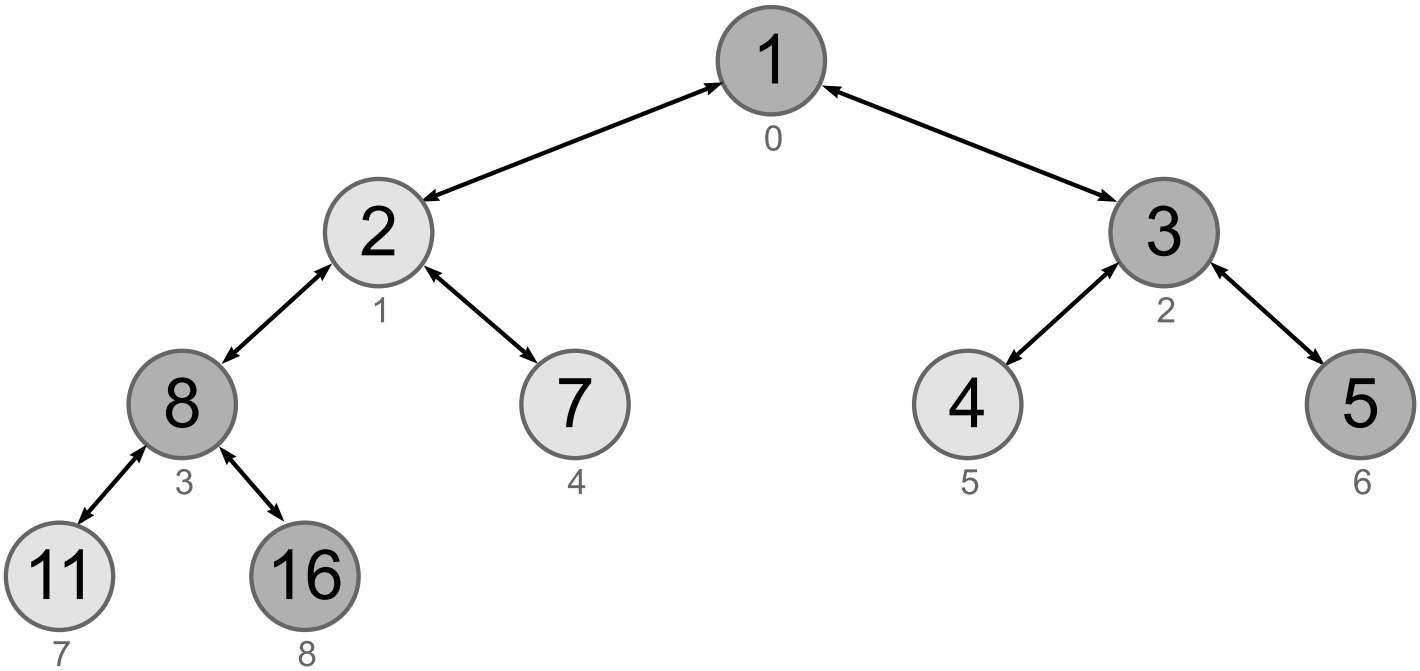
merge(pq1, pq2)

korak: 5



merge(pq1, pq2)

korak: 6



1	2	3	8	7	4	5	11	16
0	1	2	3	4	5	6	7	8

Pregled složenosti rešenja

Operacija	Niz	Binarni heap
Pravljenje praznog pq-a	$O(1)$	$O(1)$
first()	$O(1)$	$O(1)$
insert()	$O(n)$	$O(\log(n))$
extractMin()	$O(n)$	$O(\log(n))$
decreaseKey()	$O(n)$	$O(\log(n))$
delete()	$O(n)$	$O(\log(n))$
merge()	$O(n)$	$O(n)$

Binomni heap - Motivacija

- U nekim primenama želimo da efikasno spajamo dva prioritetna reda;
- Binarni heap je vrlo često dobar izbor, ali spajanje dva prioritetna reda ne radi u zadovoljavajućoj složenosti;
- Na scenu stupa **Binomni heap**, o kome će sada biti reči...

Binomno drvo

Definišimo prvo **Binomno drvo**:

- Binomno drvo *reda* 0 je samo jedan čvor (element), koji sadrži jedan Item;
- Binomno drvo *reda* k je drvo koje dobijamo kada spojimo dva binomna drveta reda $k-1$, tako što koren jednog drveta zakačimo kao najlevlje dete korenu drugog;
- Indukcijom dobijamo da binomno drvo reda k sadrži 2^k elemenata, i da koren ovakvog drveta ima tačno k dece (*red drveta*);
- Indukcijom takođe dobijamo da je *dubina* binomnog drveta reda k upravo k.

Definišimo i **stepen** binomnog drveta kao broj dece korena drveta (takođe k, zašto?).

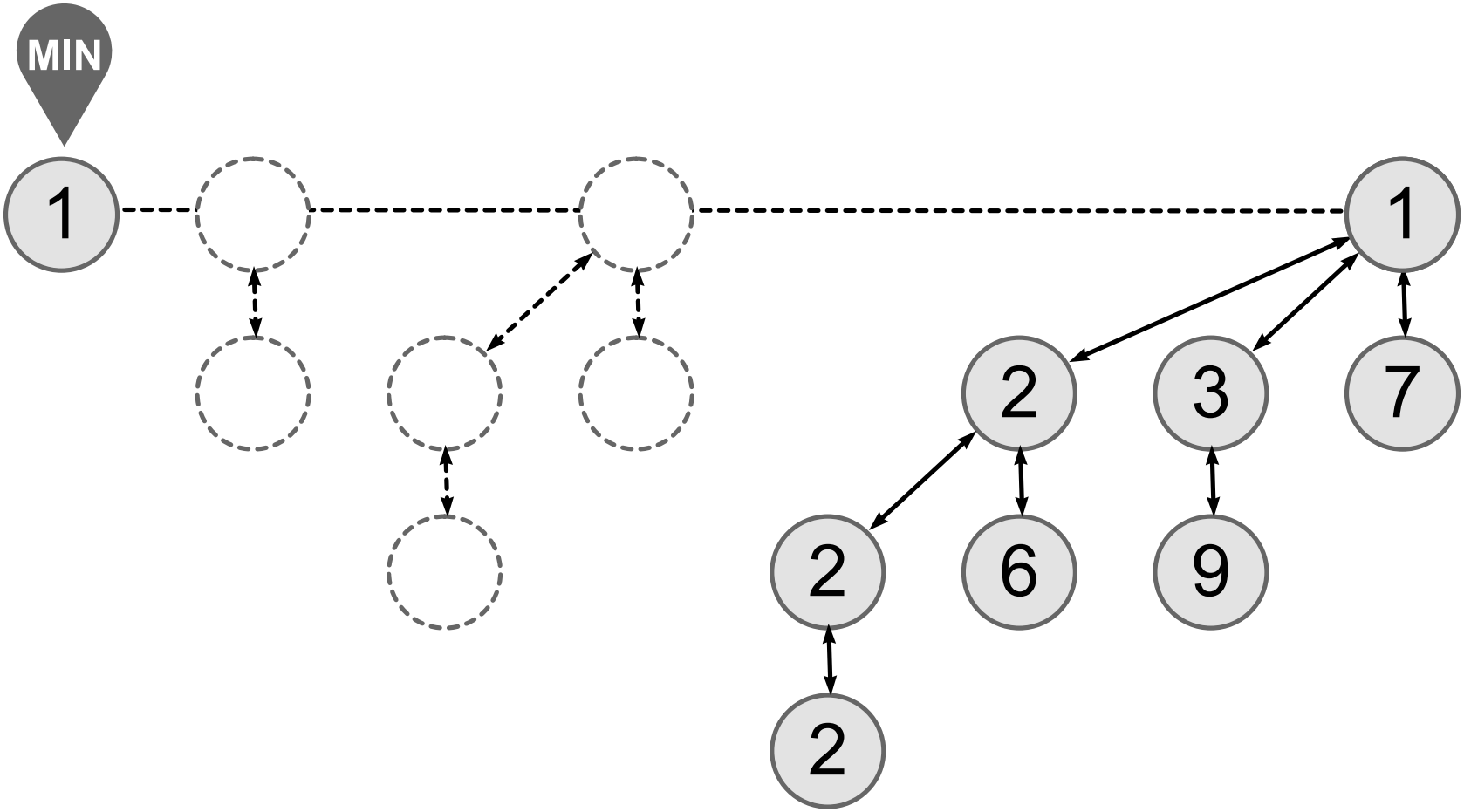
Binomni heap

Binomni heap je šuma binomnog drveća (*lista korenja*):

- Najviše po jednog za svaki red drveta (**Svojstvo jedinstvenosti**);
- Sortiranih u rastućem poretku po redovima;
- Svako drvo zadovoljava **heap svojstvo**.

NB Ukoliko binomni heap ima n elemenata, on onda sadrži $O(\log(n))$ binomnog drveća, i najveće od njih ima red $O(\log(n))$.

Primer binomnog heapa



Detalji operacija binomnog heapa

- U nastavku razmotrićemo detalje implementacije binomnog heapa;
- Uvedimo prvo novu strukturu podataka, **povezanu listu**...

Povezana (ulančana) lista

- Podaci se nalaze u formi liste elemenata;
- Svaki element sadrži podatak (Item) i informaciju o poziciji sledećeg elementa;
- Poslednji element u listi pokazuje na "nepostojeći" element (NULL), koji označava kraj liste;

Povezana (ulančana) lista, cont'd

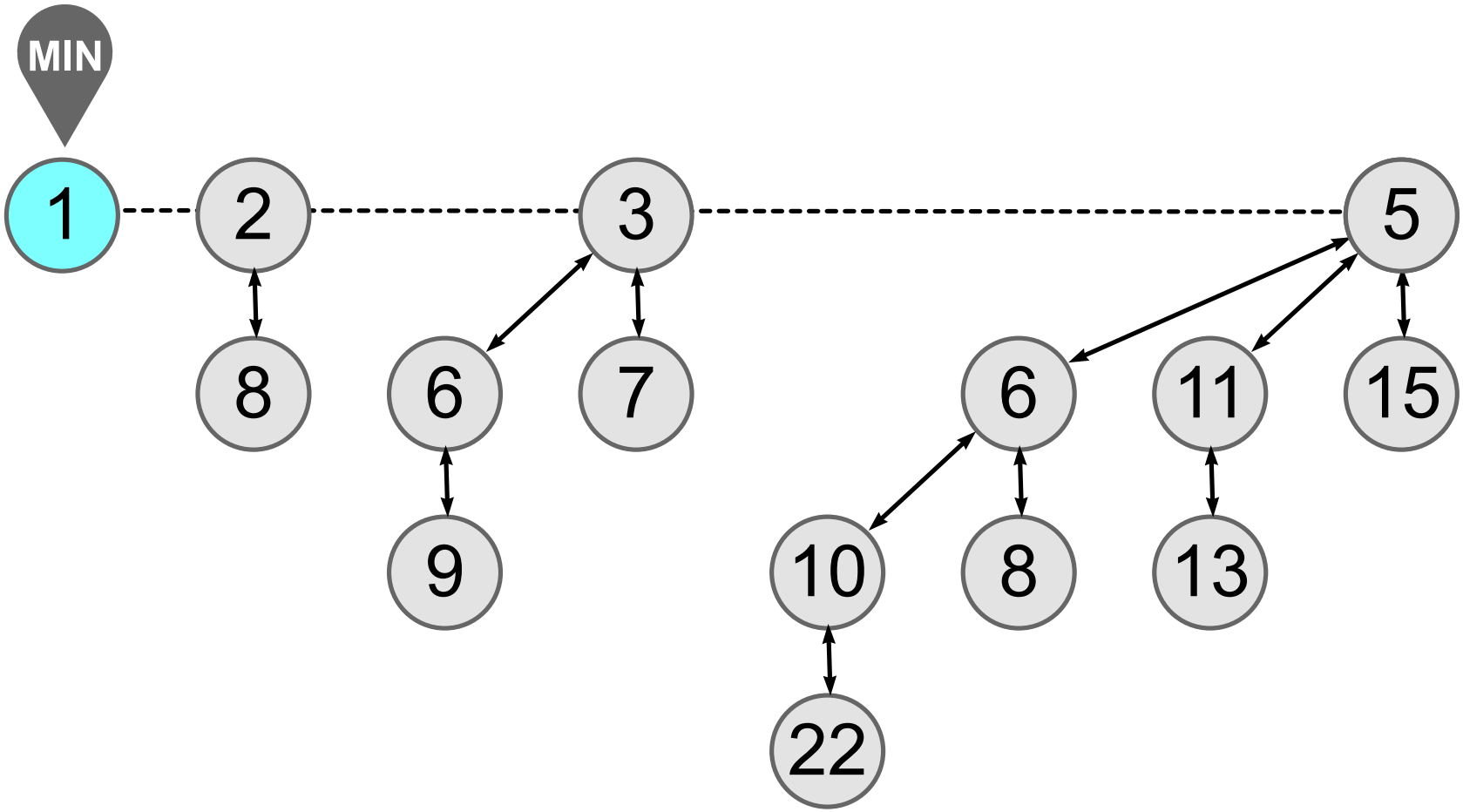
- Ovakva struktura omogućava prosto uklanjanje i dodavanje novih elemenata na proizvoljnom mestu u listi (u konstantnoj složenosti);
- Može biti i **dvostruko ulančana**, ukoliko svaki element pokazuje i na element koji mu prethodi u listi;
- Mana ove strukutre je linearna složenost pretraživanja.

NB Listu korenja u memoriji čuvamo u formi dvostruko povezane liste.

first()

- Prva ideja: Pretražimo listu korenja u $O(\log(n))$;
- Možemo i bolje, uz malo memorije!
- Druga ideja: Čuvamo pokazivač na najmanji koren u listi korenja, i korigujemo ga ukoliko je to potrebno. Vreme potrebno: $O(1)$.

first()
korak: 1



merge()

- Spajanje dva binomna heapa vršimo na način sličan sabiranju dva binarna broja.
- Možemo postaviti sledeću analogiju:
  - odsustvo binomnog drveta reda $k \Leftrightarrow$ binarno 0
  - prisustvo binomnog drveta reda $k \Leftrightarrow$ binarno 1

  Na poziciji k u zapisu binarnog broja koji odgovara jednom binomnom heapu;
- Binarno sabiranje na k-toj poziciji binarnog broja:
  - $0 + 0 \Leftrightarrow$ odsustvo drveta reda k
  - $1 + 0 \Leftrightarrow$ prisustvo drveta reda k
  - $1 + 1 \Leftrightarrow$ odsustvo drveta reda k i novo drvo reda $k + 1$ (*carry*)

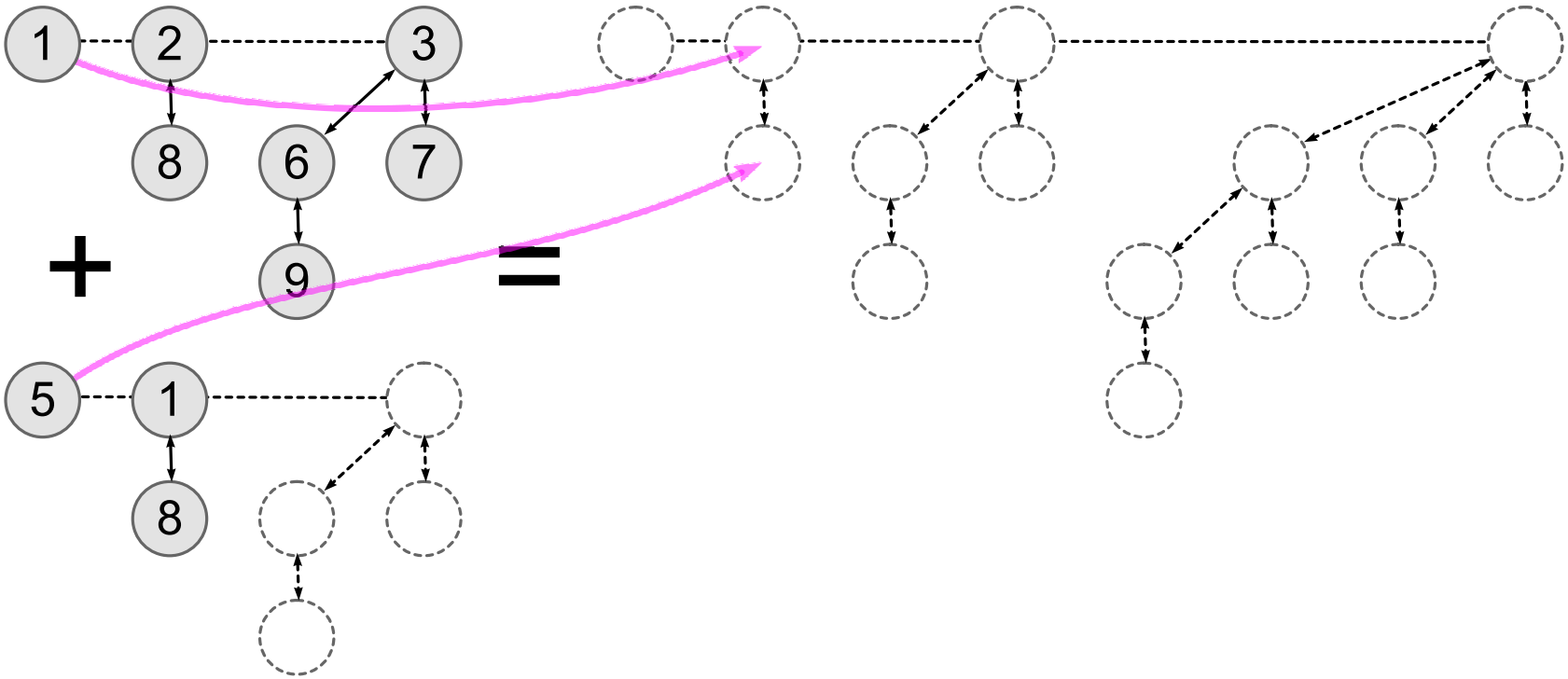
  U konačnom binomnom heapu.

merge(), cont'd

- Dakle, postupak kojim spajamo dva binomna heapa je sledeći:
  - Počevši od drveta reda 0 "sabiramo" drveće odgovarajućeg reda pazeći na carry:
    - Ukoliko ne postoji ni jedno drvo za određeni nivo (računajući carry), nastavljamo na sledeći nivo;
    - Ukoliko postoji jedno drvo za određeni red, dodajemo ga u konačan heap i nastavljamo na sledeći nivo;
    - Ukoliko postoje dva drveta za određeni red, spajamo ih u drvo višeg reda i njega računamo kao carry za sledeće sabiranje;
    - Ukoliko postoje tri drveta za određeni nivo, jedno dodajemo u konačan heap, a druga dva spajamo u drvo višeg reda i računamo kao carry za sledeće sabiranje;
  - Kada "istrošimo" sve redove, dodajemo eventualy carry u konačan heap i završili smo.

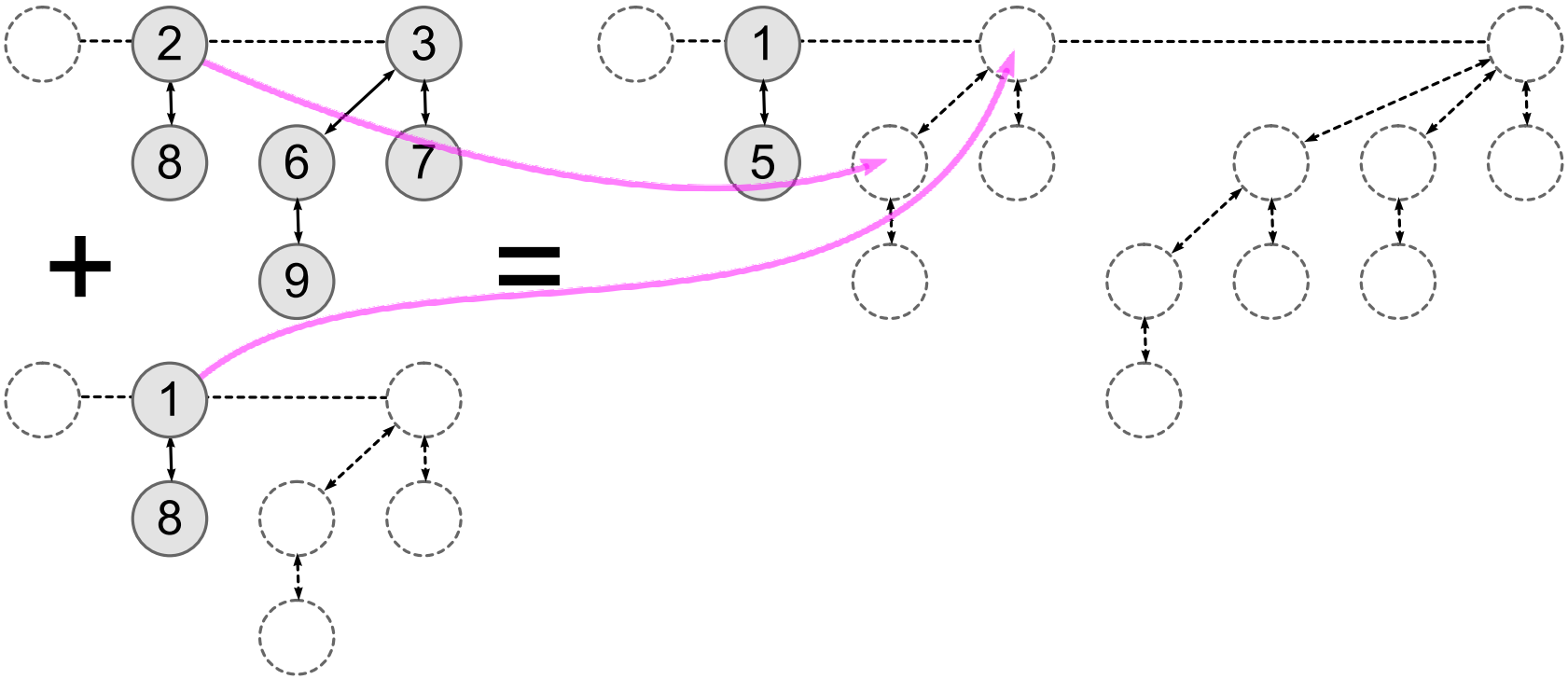
merge(pq1, pq2)

korak: 1



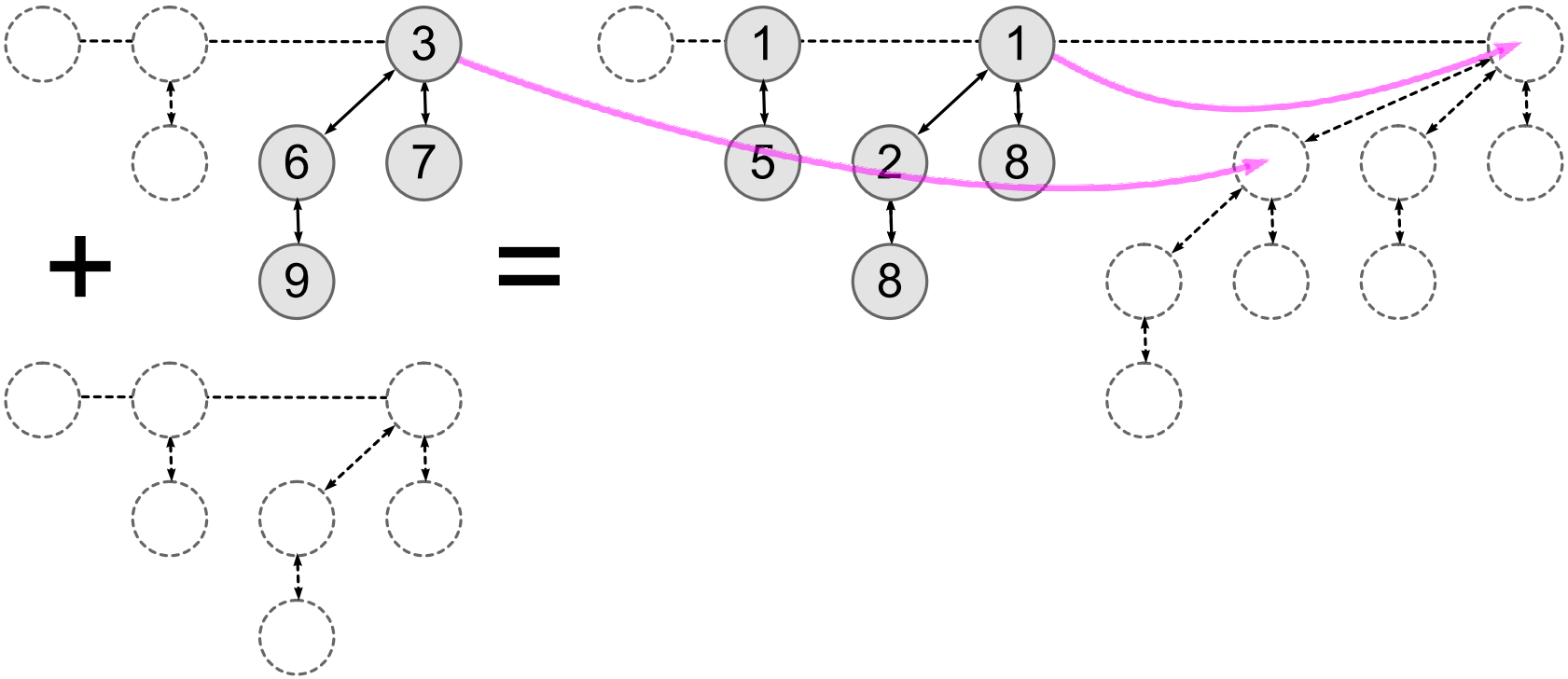
merge(pq1, pq2)

korak: 2



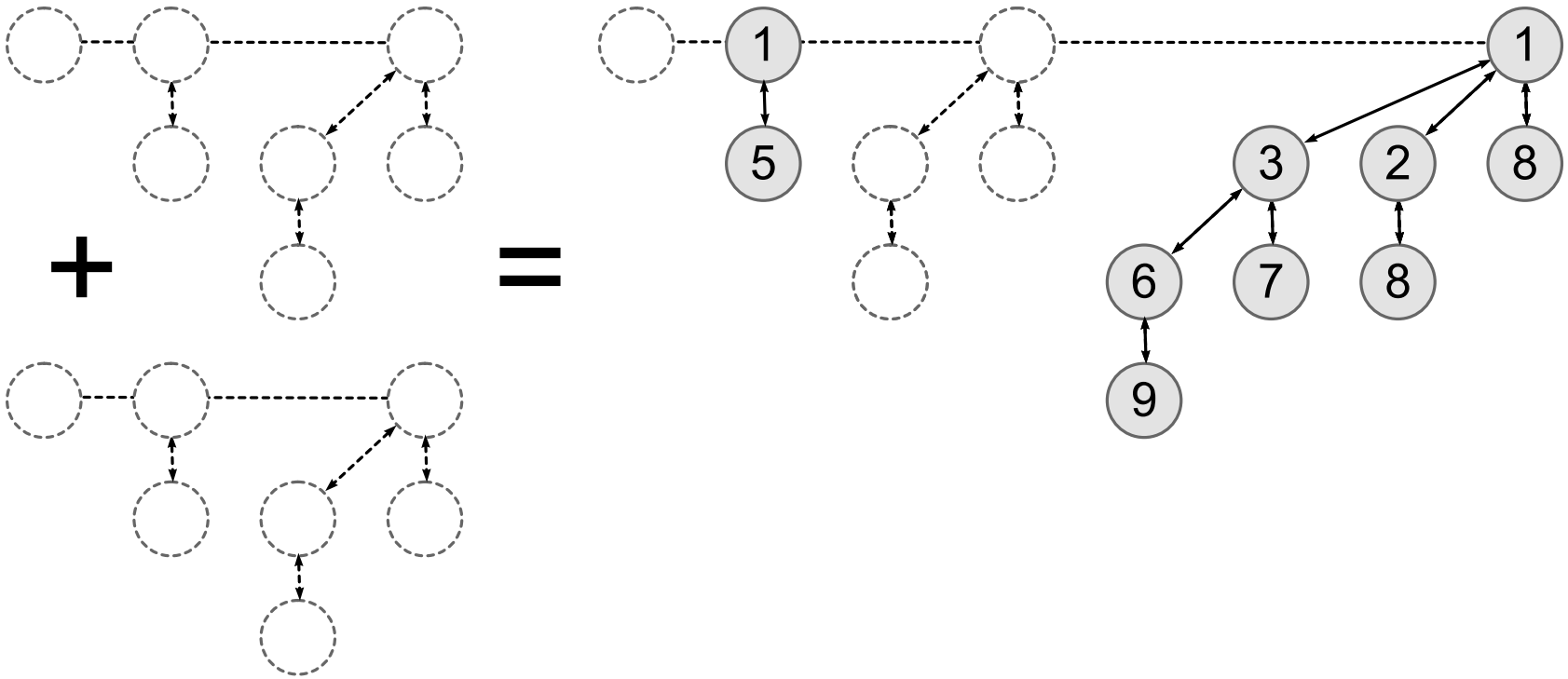
merge(pq1, pq2)

korak: 3



merge(pq1, pq2)

korak: 4



extractMin()

- Deca minimalnog čvora čine novu listu korenja, odnosno binomni heap;
- Dakle, kada izbacimo minimalni element, kreiramo novi binomni heap od njegove dece;
- Novi heap merge-ujemo sa početnim;
- Jednim prolaskom kroz listu korenja konačnog heapa nađemo novi minimum i korigujemo pokazivač;
- Pošto svaka od ovih operacija radi u $O(\log(n))$, konačna složenost je $O(\log(n))$.

insert()

- Element koji dodajemo posmatramo kao binomno drvo reda 0;
- Izvedemo merge nad "jediničnim" binomnim heapom koji sadrži novi element i originalnim heapom;
- Pošto merge radi u $O(\log(n))$, konačna složenost je $O(\log(n))$.

decreaseKey()

- Postupak je isti kao i kod binarnog heapa;
- Dokle god je element, čiji je ključ smanjen, manji od svog roditelja u odgovarajućem binomnom drvetu, menjamo ga sa roditeljem;
- Pošto je dubina binomnog drveća ograničena sa $O(\log(n))$, konačna složenost je $O(\log(n))$.

Pregled složenosti rešenja

Operacija	Niz	Binarni heap	Binomni heap
Pravljenje praznog pq-a	$O(1)$	$O(1)$	$O(1)$
first()	$O(1)$	$O(1)$	$O(1)$
insert()	$O(n)$	$O(\log(n))$	$O(\log(n))$
extractMin()	$O(n)$	$O(\log(n))$	$O(\log(n))$
decreaseKey()	$O(n)$	$O(\log(n))$	$O(\log(n))$
delete()	$O(n)$	$O(\log(n))$	$O(\log(n))$
merge()	$O(n)$	$O(n)$	$O(\log(n))$

Fibonacci heap - Motivacija

- Iako je binomni heap poboljšao binarni na polju spajanja dva prioritetna reda, možemo da postignemo još neverovatnije složenosti ukoliko binomni heap "ulenjimo", i malo modifikujemo;
- Na ovoj ideji bazira se **Fibonacci heap**;
- Ova struktura podataka obećava **konstantnu** složenost za sve operacije sem extractMin() i delete(), uz napomenu da je ta konstanta veoma velika!

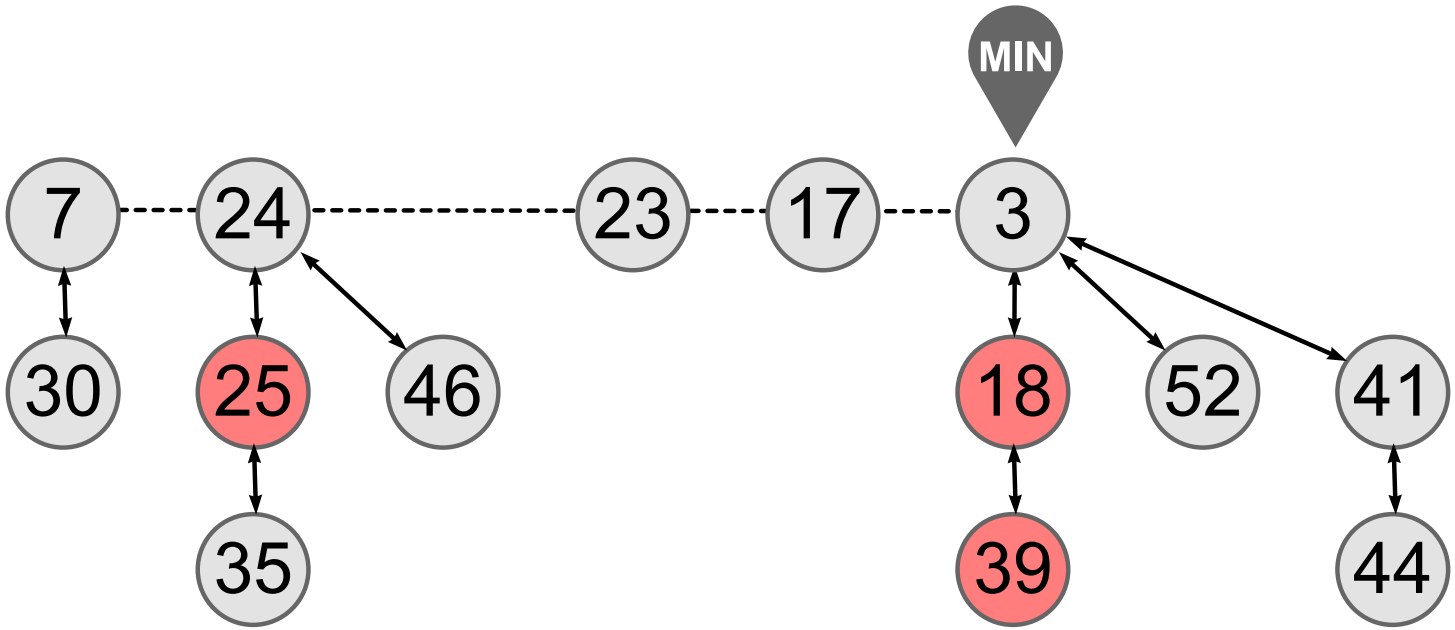
Fibonacci heap - Definicija

Fibonacci heap je struktura podataka koja ima sledeća, bitna svojstva:

- Prvenstveno je **lenja** struktura, jer većina operacija učini ono najmanje što mora, bez ikakvog naknadnog uređivanja i održavanja strukture, za razliku od Binomnog heapa(insert() i merge());
- Sastoji se od kolekcije drveća, gde svako drvo zadovoljava heap svojstvo;

Pogledajmo sada detalje implementacije jednog Fibonacci heapa...

Primer Fibonacci heapa

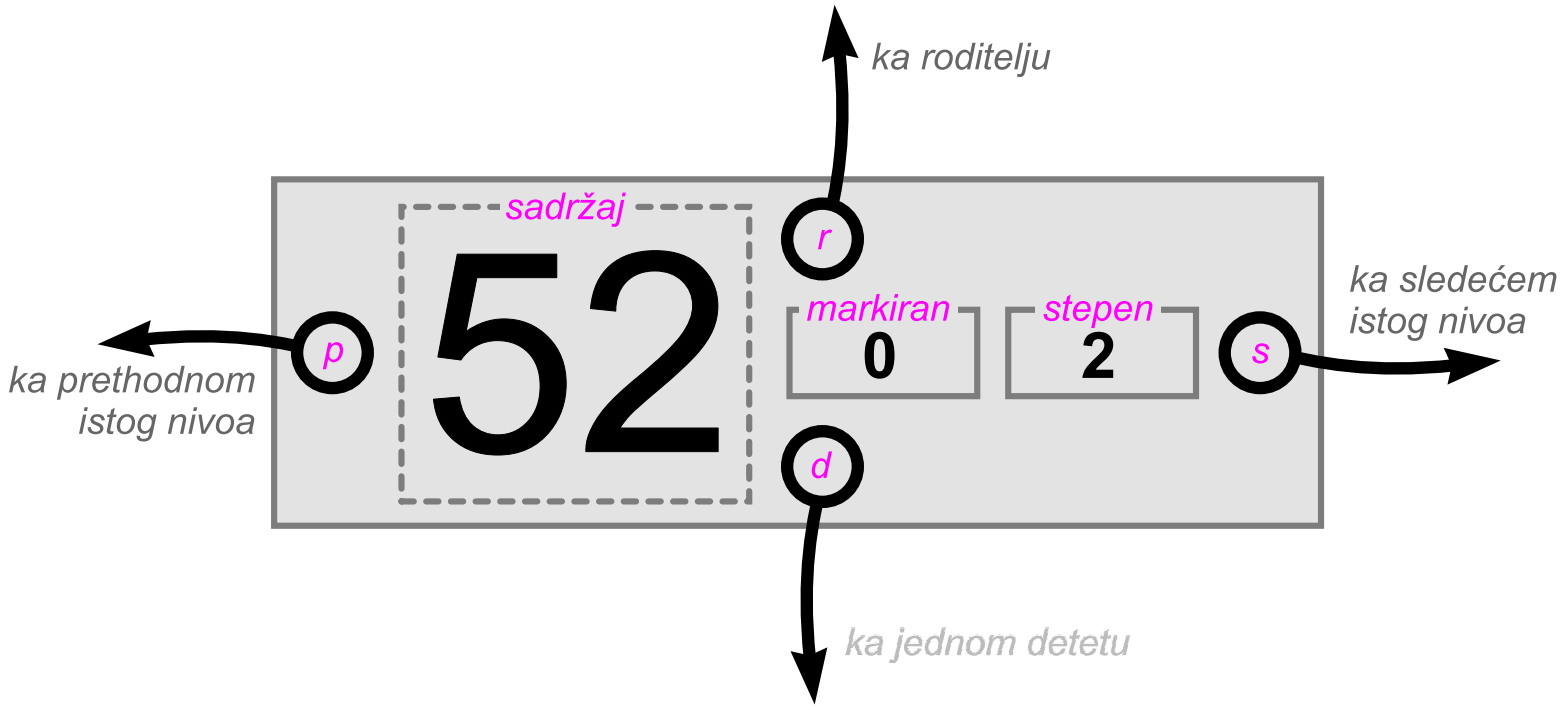


Detalji implementacije

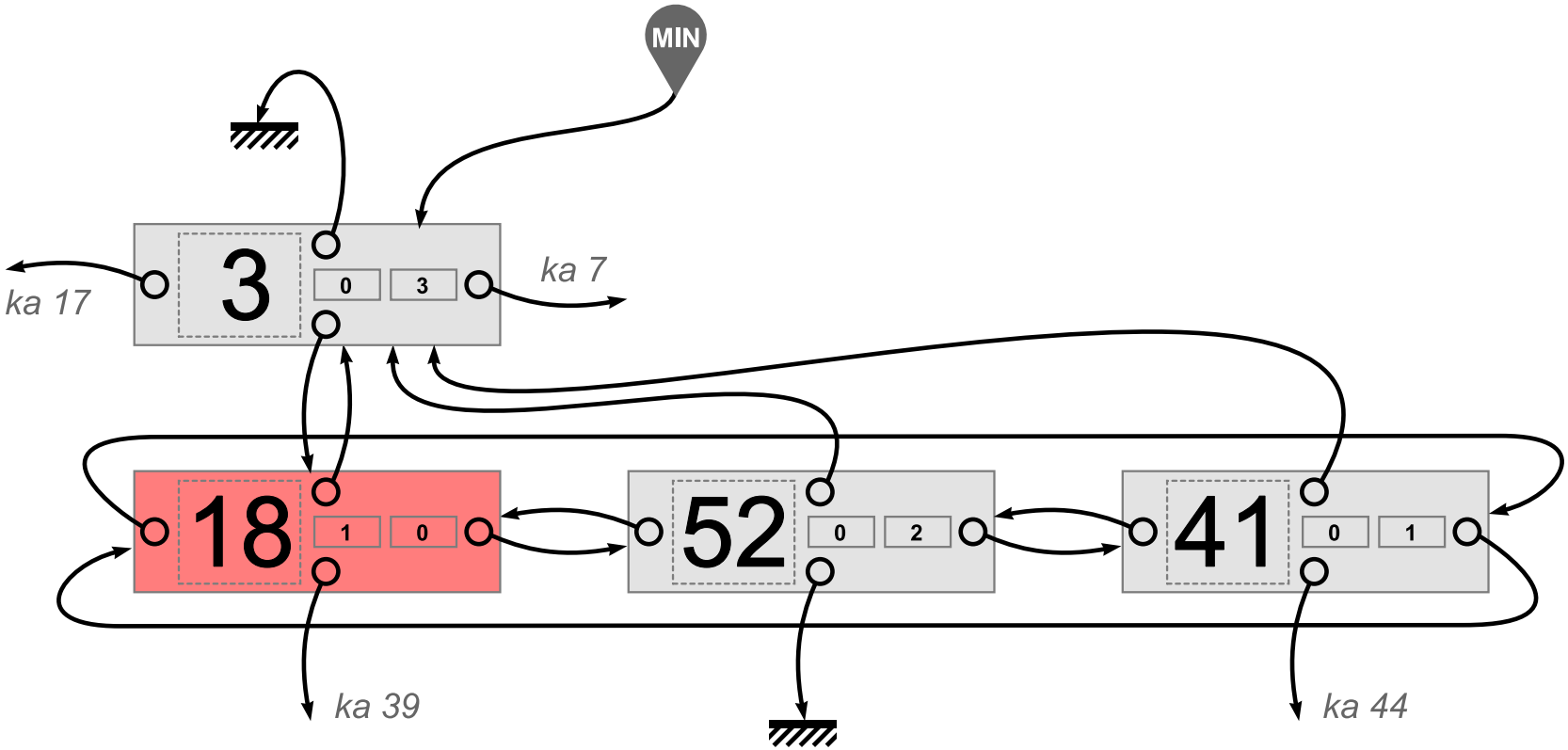
Osnovni gradivni elementi Fibonacci heapa su:

- Dvostruko ulančana, ciklična lista korenja drveća, gde svako drvo ima sledeće osobine i polja:
  - Zadovoljava heap svojstvo;
  - Deca svakog čvora su takođe povezana u dvostruko ulančanu, cikličnu listu, bez posebnog redosleda;
  - Svaki čvor sadrži pokazivač na roditelja, kao i na neko svoje dete (NULL ukoliko nema dece);
  - Polje **markiran** koje naznačava da li je čvor obeležen (više reči o ovome kasnije);
  - Polje stepen;
- Pokazivač na koren sa sadržajem najvećeg značaja (*min pointer*).

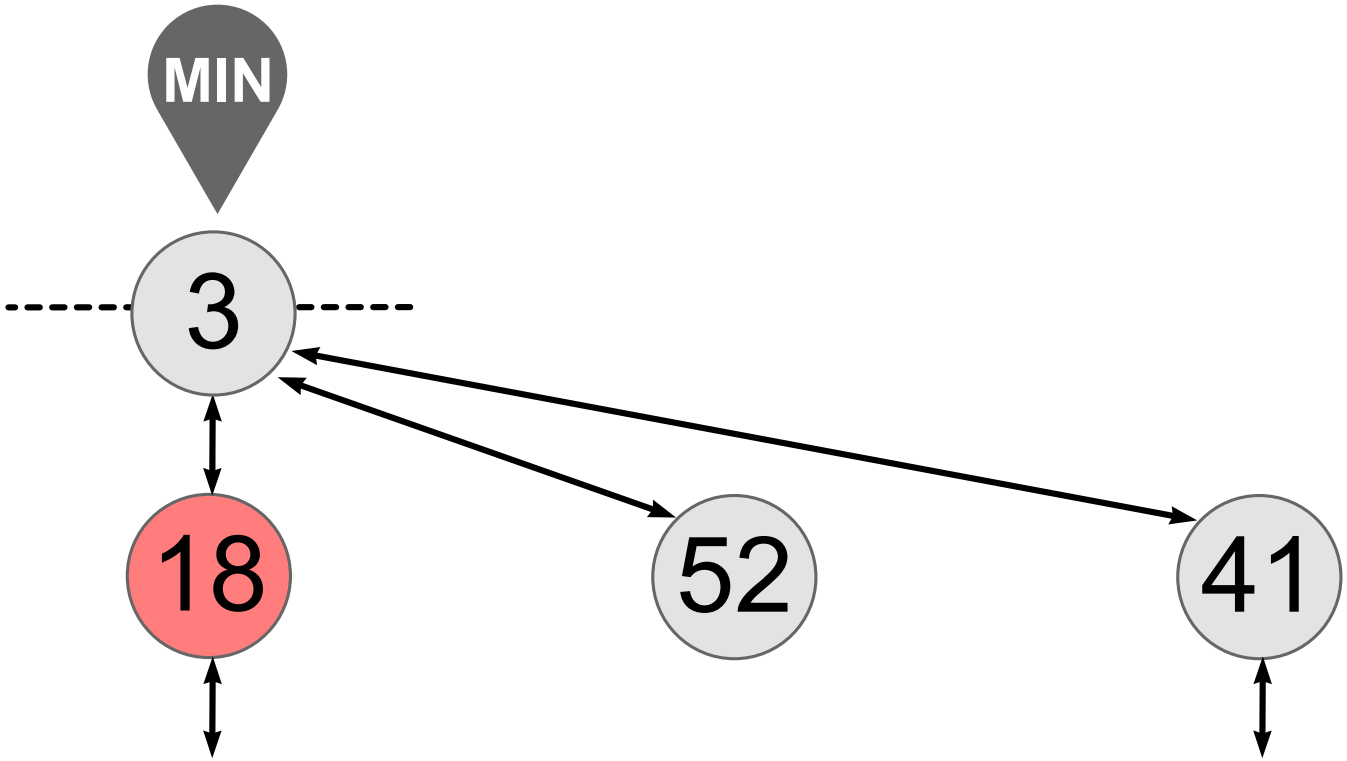
Primer: Fib-čvor



Primer: Fib-drvo struktura



Primer: Fib-drvo reprezentacija



Markiranje

Neobičan uslov koji svi Fib-čvorovi moraju da zadovolje, pored heap svojstva, je sledeći:

- Nakon što neki čvor postane dete drugog čvora, taj čvor može izgubiti **najviše jedno** svoje dete pre nego što mora ponovo da postane koren;
- Primer: čvor 3 (koreni) i čvor 43 (ne-koreni) u prethodnom primeru;
- Više o ovom uslovu reći ćemo kada se budemo bavili analizom složenosti extractMin();
- Realizacija: Kada nekom ne-korenom Fib-čvoru odsečemo dete **markiramo** ga ukoliko nije bio markiran, a promovišemo ga u koren ukoliko jeste.

Implementacija metoda - komentari

- Kao i ostali heapovi, ni Fibonacci heap nema efikasnu metodu za pronalaženje proizvoljnog čvora u strukturi;
- Zato, sve metode koje ćemo navesti, kao ulaz (ukoliko ga imaju) uzimaju već pripremljene Fib-čvorove i vraćaju Fib-čvorove (gde je to potrebno);
- Pogledajmo sada tehnike implementacije svake metoda Fibnacci heapa...

constructor()

Kreira Fibonacci heap sa jednim čvorom, tako što:

- Postavi min-pointer na taj čvor;
- Taj čvor "poveže samog sa sobom".

Složenost: $O(1)$

first()

Vraća pokazivač na minimalni element, tako što:

- Vrati sadžaj min pointera.

Složenost: $O(1)$

merge()

Spaja dva Fibonacci heapa tako što:

- "Preveže" liste korenja u novu dvostruko povezanu listu korenja (uništava početne heapove);
- Postavi novi min pointer na manji od min pointera prvog i drugog početnog heapa.

Složenost: $O(1)$

insert()

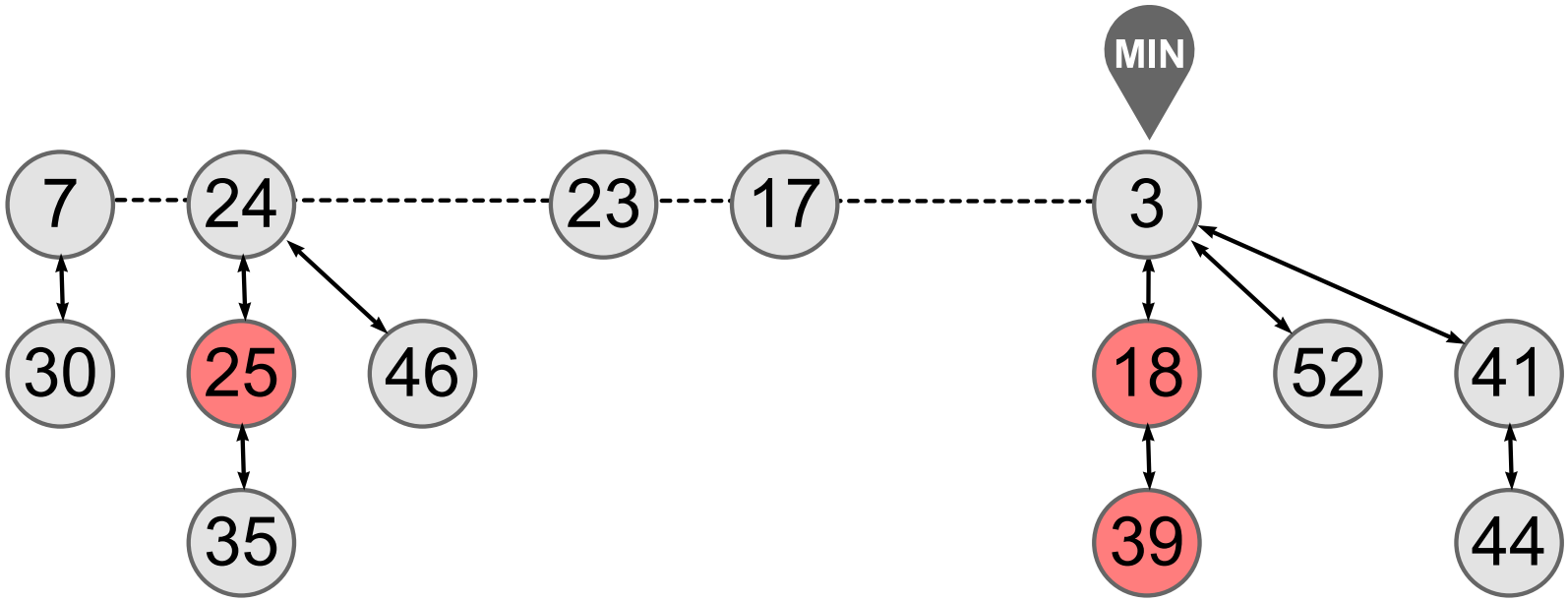
Dodaje novi čvor u heap, tako što:

- Napravi Fibonacci heap sa jednim čvorom (constructor());
- Pozove merge() za jedinični heap i polazni heap.

Složenost: $O(1)$

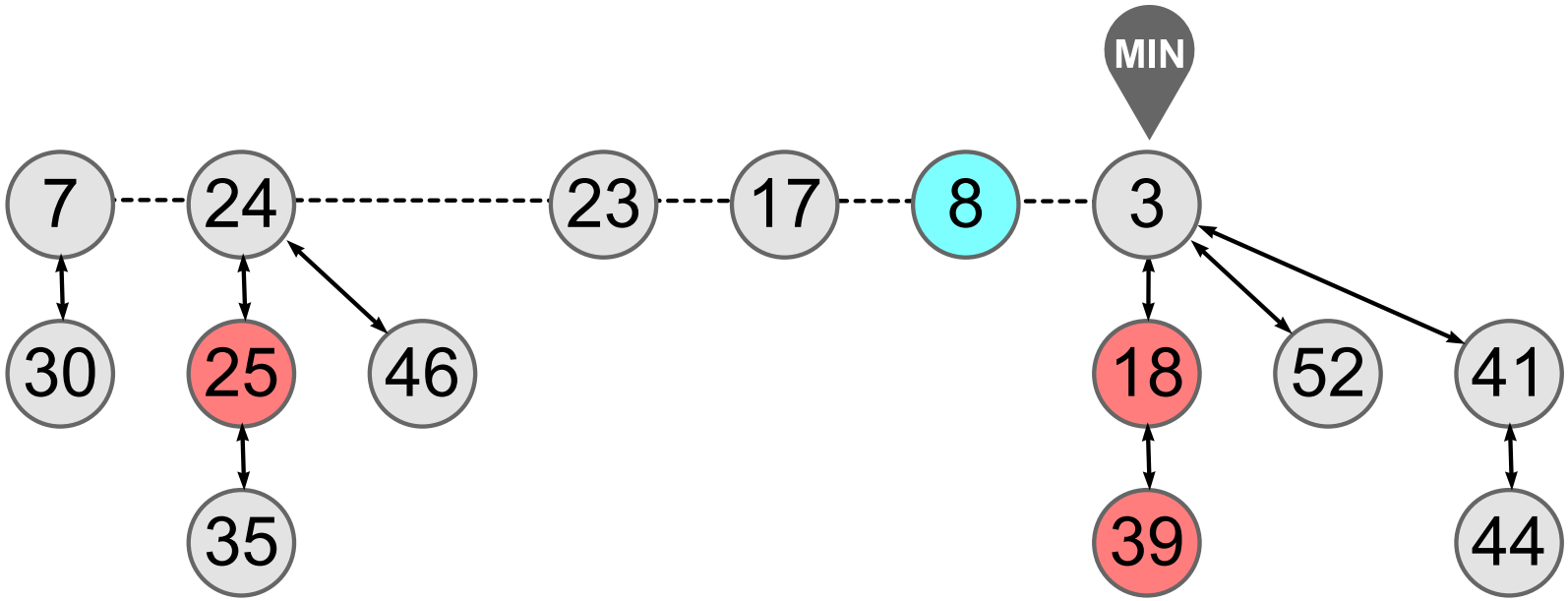
insert(8)

korak: 1



insert(8)

korak: 2



extractMin()

Izbacuje minimalni Fib-čvor iz heapa, tako što:

- Iseče minimalni Fib-čvor iz korene liste ($O(1)$);
- Odseče minimalni Fib-čvor od njegove dece (cena proporcionalna broju dece);
- Merge-uje listu dece minimalnog Fib-čvora sa početnom listom korenja ($O(1)$);

extractMin(), cont'd

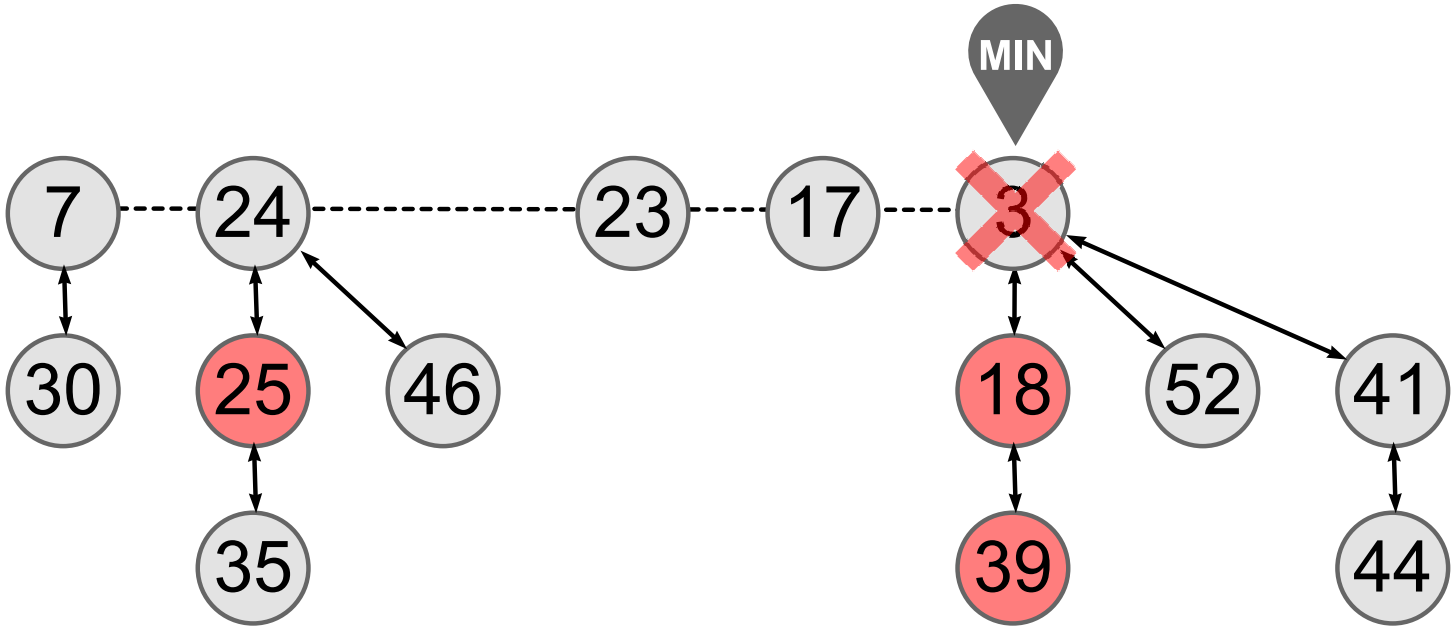
- Koriguje novu listu korenja na sledeći način:
  - Ukoliko dva drveta u listi korenja imaju isti stepen k (broj dece korena), spaja ih u novo drvo stepena $k+1$, tako što drvo sa većim korenom doda u listu dece drveta sa manjim korenom (cenu ćemo diskutovati kasnije);
  - Ponavlja prvi korak dok heap ne zadovolji uslov jedinstvenosti (kao kod binomnog heapa);
- Koriguje min pointer nakon prolaska kroz konačnu listu korenja (cenu diskutujemo kasnije);
- Vrati minimalni Fib-čvor koji je isečen na početku.

Složenost: $O(\log(n))^*$

***nije worst-case!**

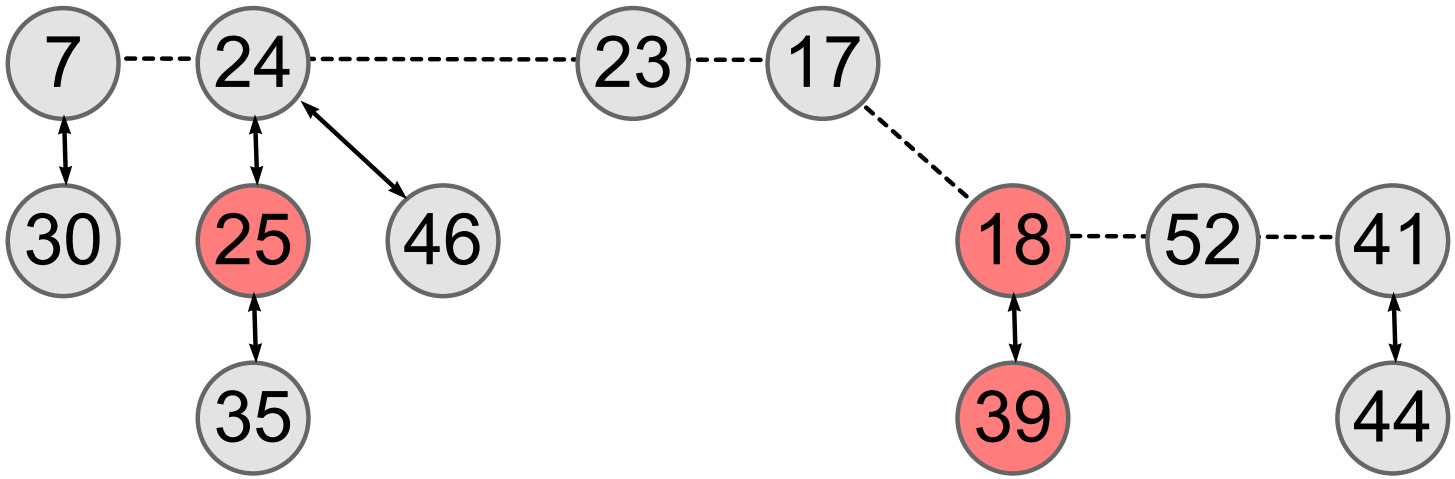
extractMin()

korak: 1



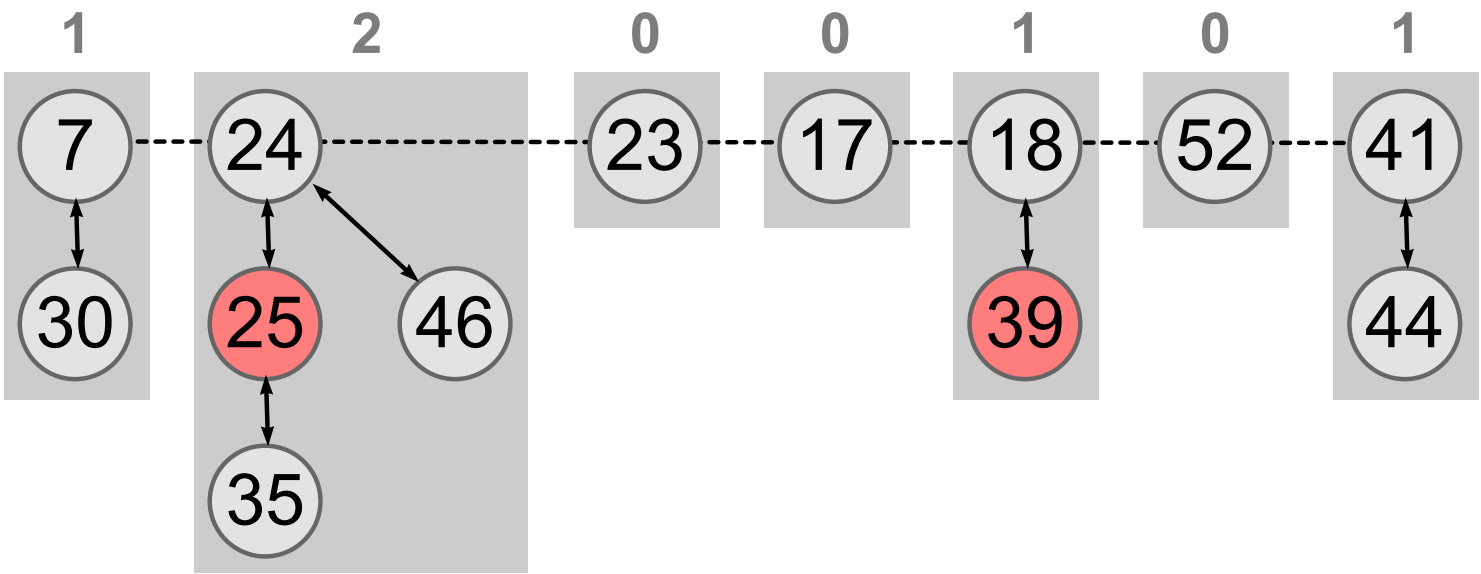
extractMin()

korak: 2



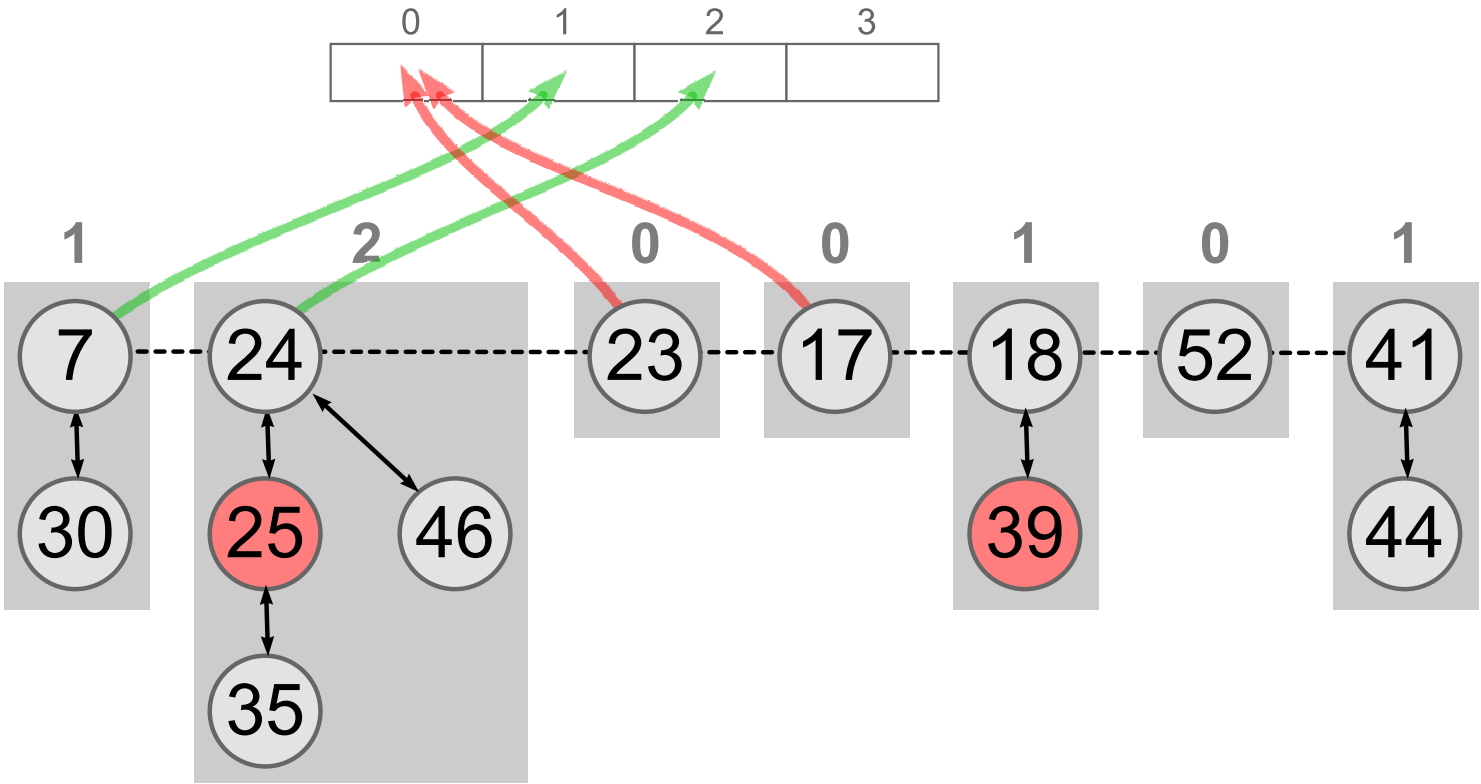
extractMin()

korak: 3



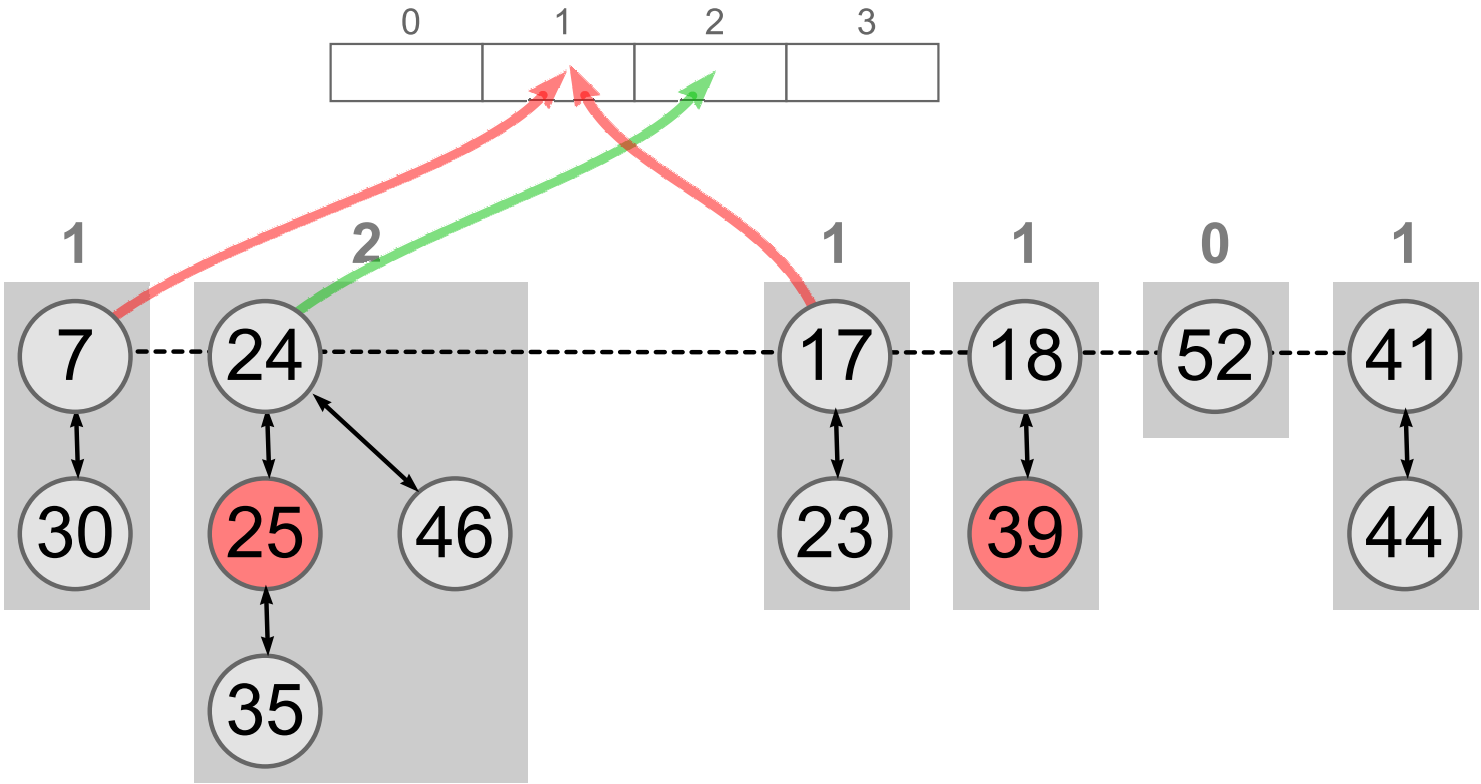
extractMin()

korak: 4



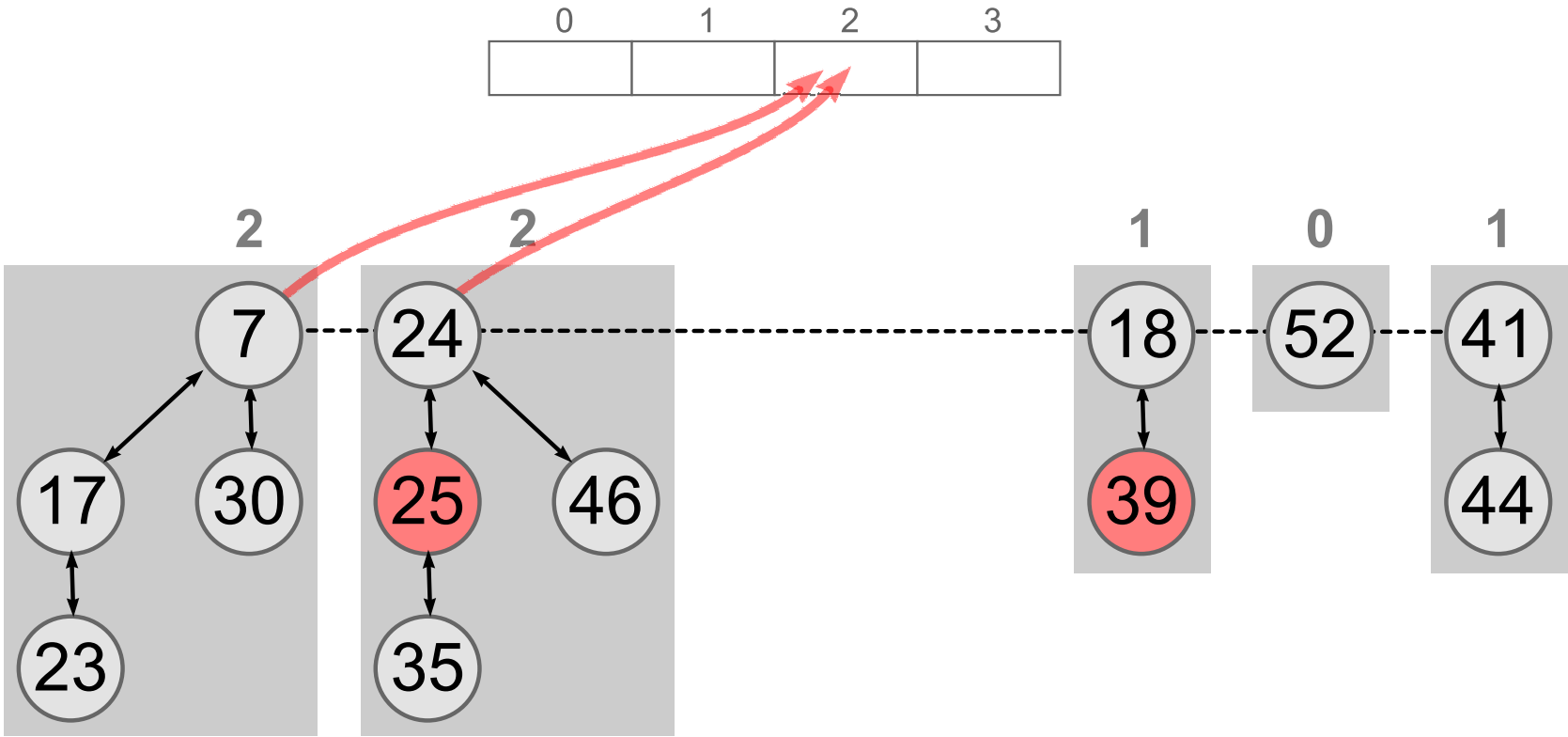
extractMin()

korak: 5



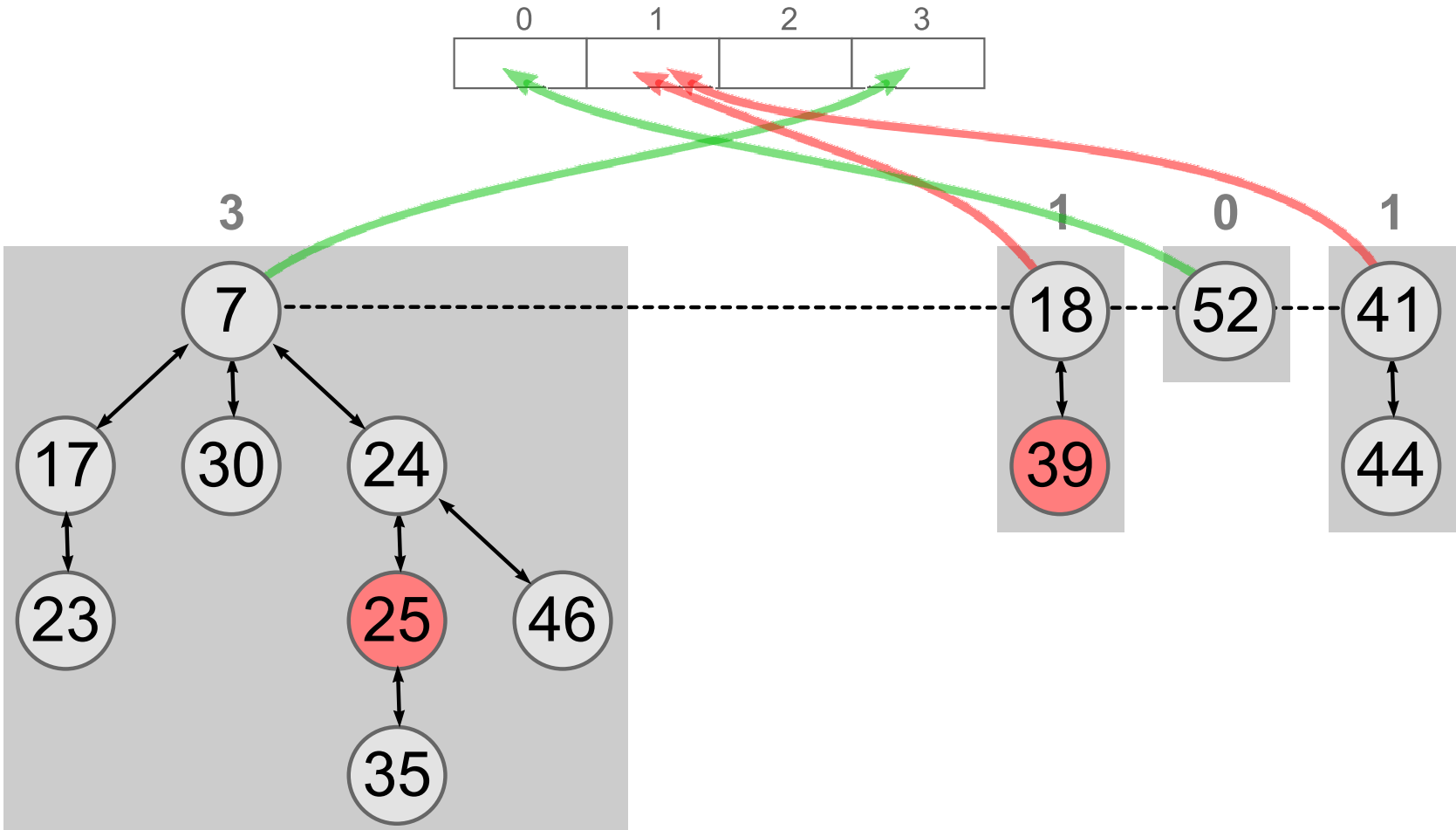
extractMin()

korak: 6



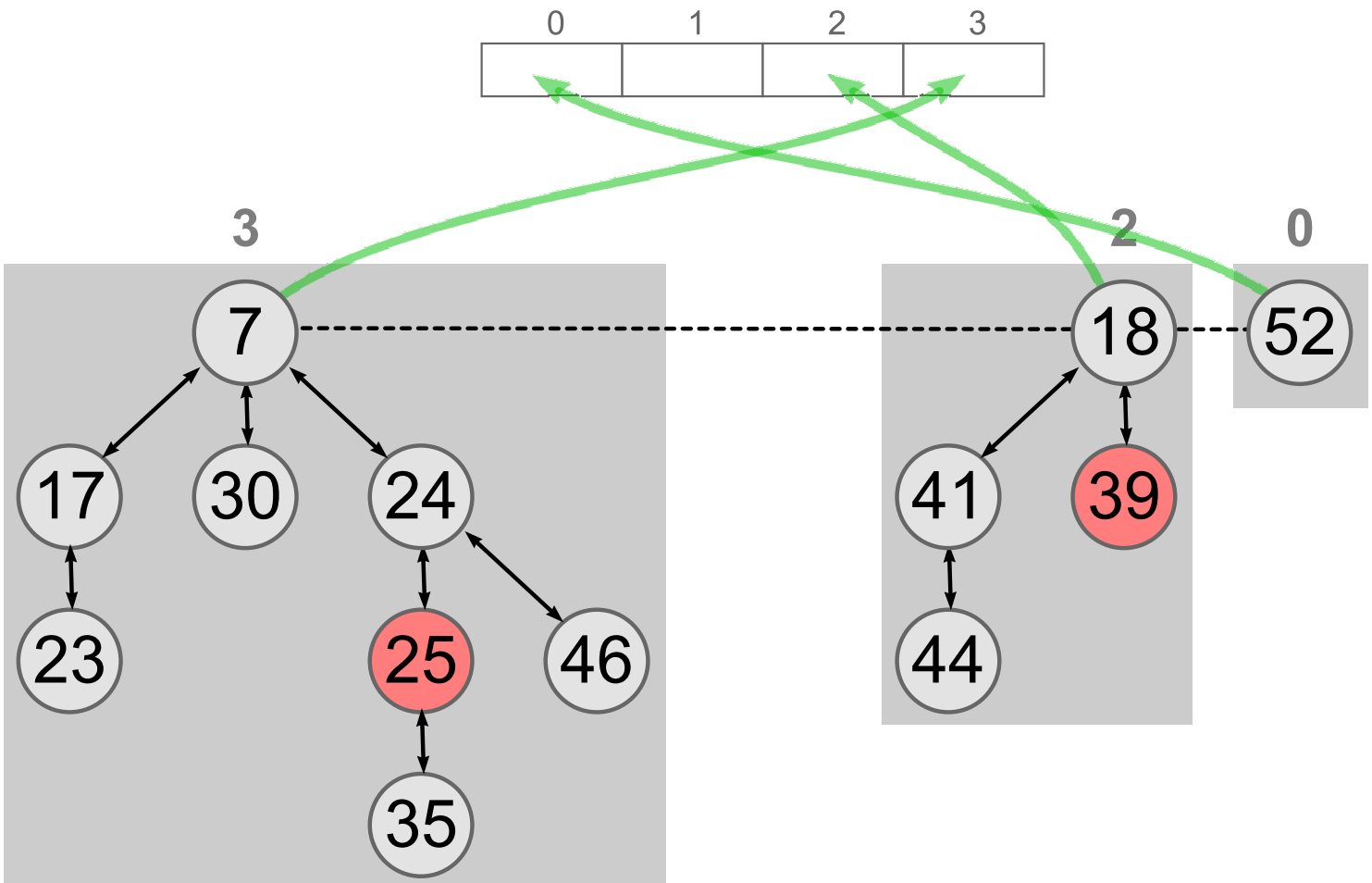
extractMin()

korak: 7



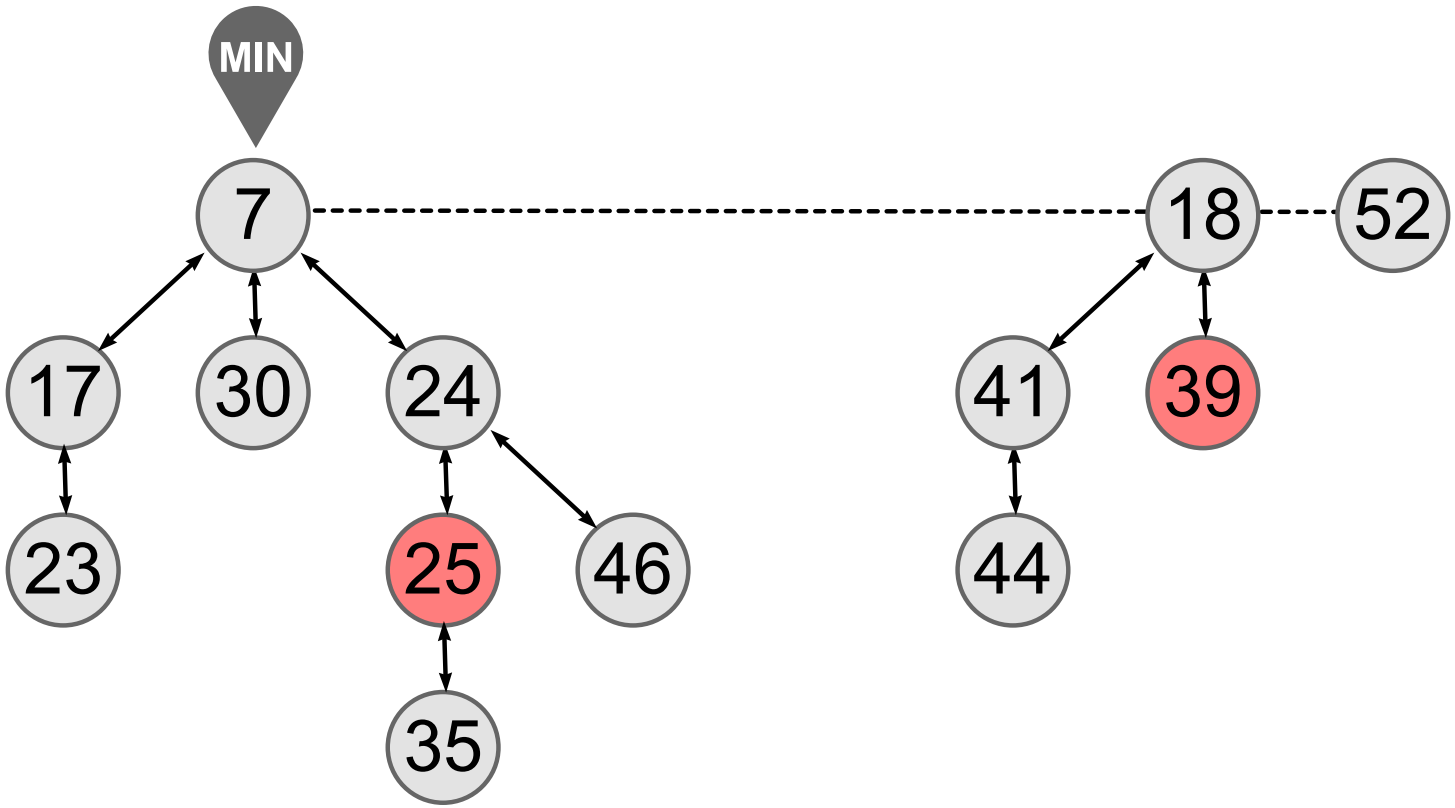
extractMin()

korak: 8



extractMin()

korak: 9



decreaseKey()

Smanjuje ključ elementa (nazovimo ga **n**) u Fibonacci heapu, tako što:

- Smanji vrednost ključa n-a na datu vrednost;
- Proveri da li je novi ključ sada manji od ključa roditelja elementa n:
  - Ukoliko **nije**, sve ostaje na svome mestu i završava;
  - Ukoliko **jeste**, heap svojstvo više nije zadovoljeno!

decreaseKey(), cont'd

Heap svojstvo dalje koriguje tako što:

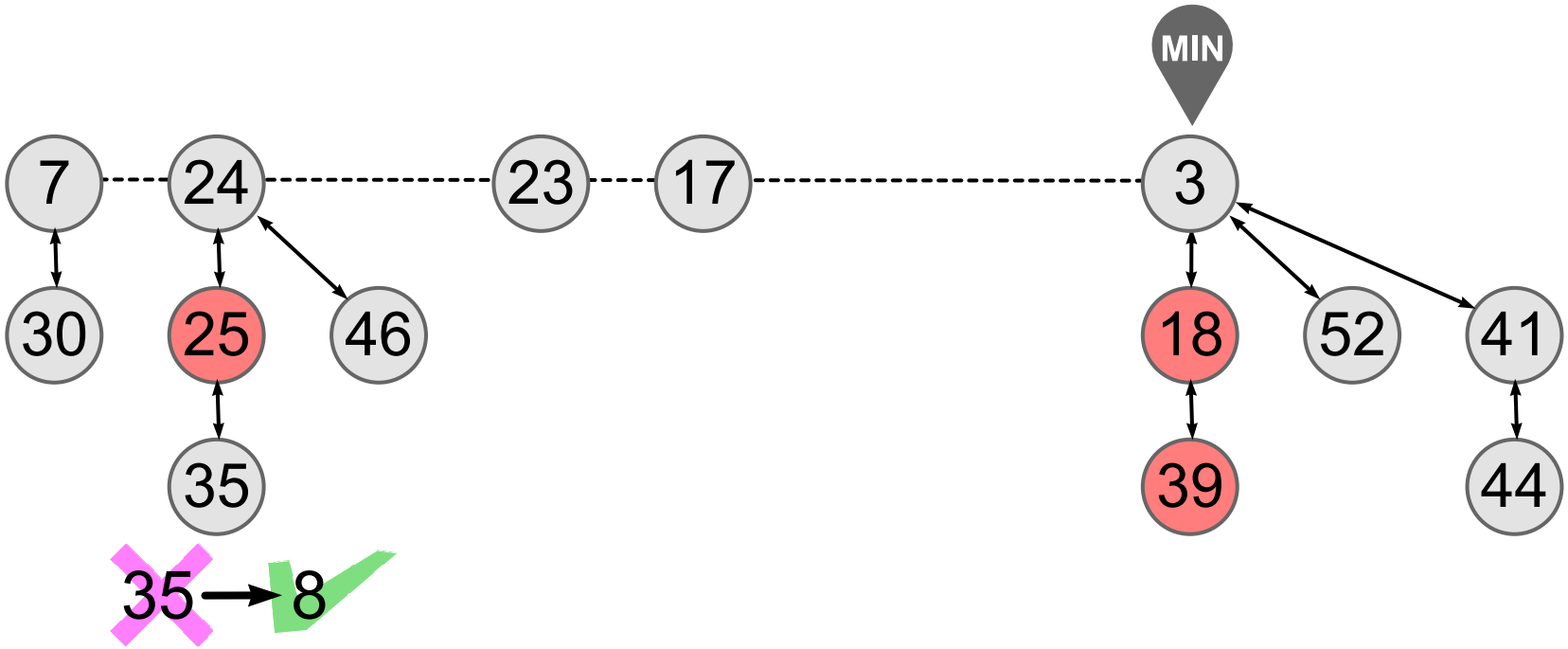
- Iseče element n i odmarkira ga (ukoliko je bio markiran);
- Pozove merge za n i listu korenja;
- Proveri da li je bivši roditelj (ukoliko postoji) n-a bio markiran:
  - Ukoliko **nije**, završava;
  - Ukoliko **jeste**, roditelj postaje n, i vraćamo se na prvi korak sa ovog slajda;

NB Ovakvo iterativno isecanje markiranih čvorova naziva se **kaskadni rez**.

Složenost: $O(1)^*$

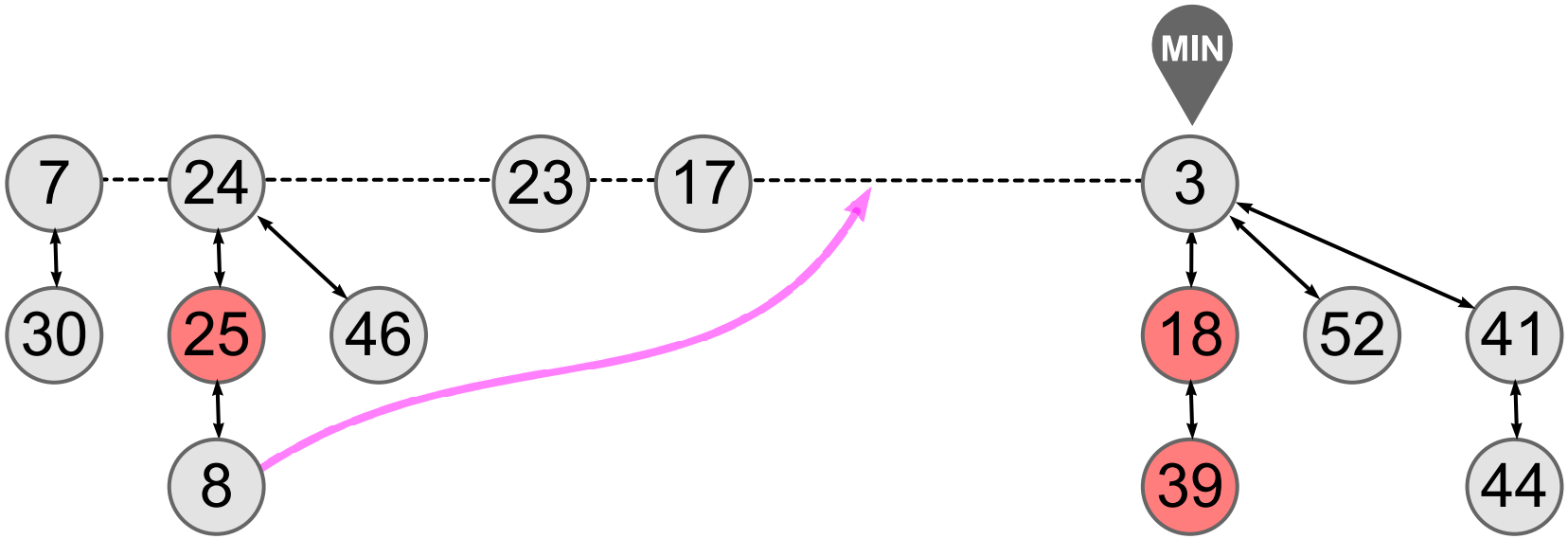
decreaseKey(35, 8)

korak: 1



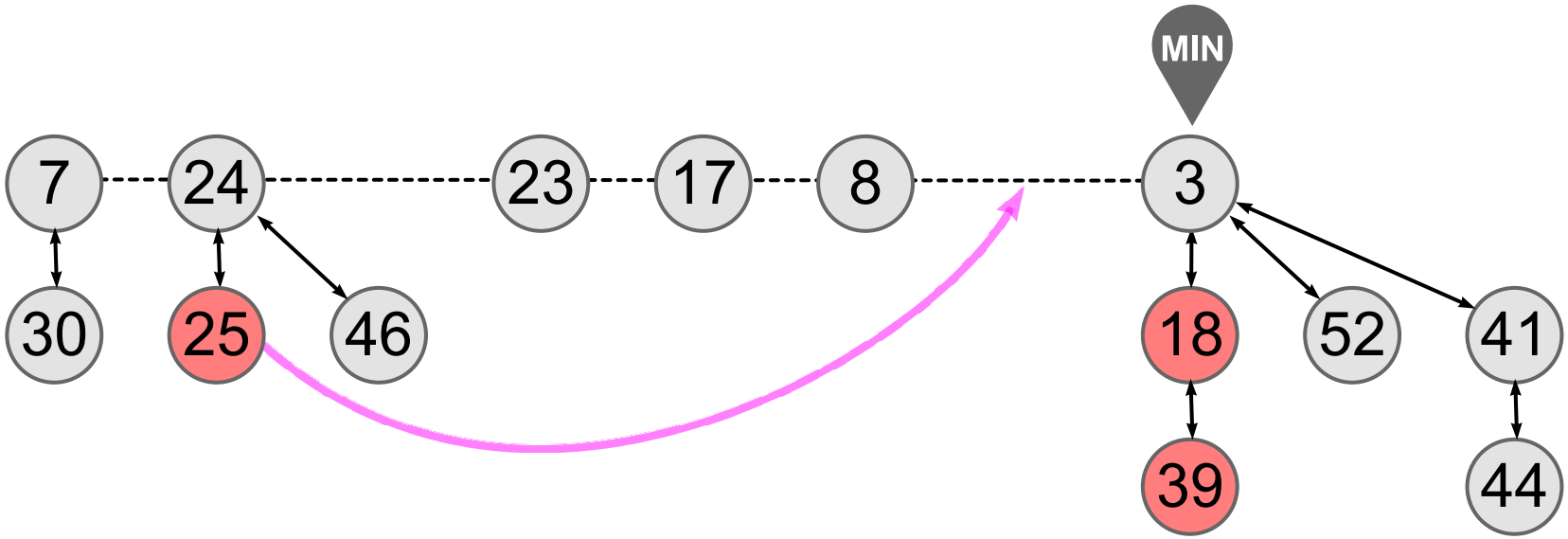
decreaseKey(35, 8)

korak: 2



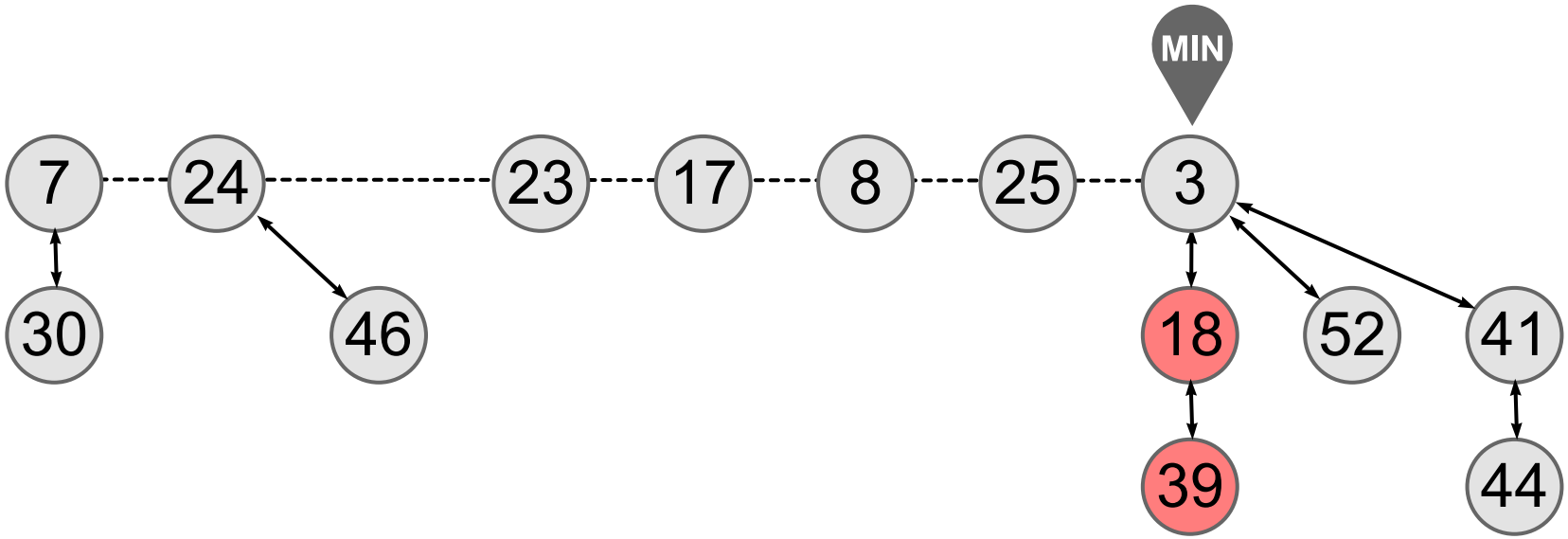
decreaseKey(35, 8)

korak: 3



decreaseKey(35, 8)

korak: 4



delete()

Uklanja element iz Fibonacci heapa, tako što:

- Pozove decereaseKey() na taj element i postavi mu ključ na $-\infty$
- Pozove extractMin() za ceo heap.

Složenost: $O(\log(n)^*)$

Uvod - Bolja procena

- Da bismo dokazali asimptotske složenosti Fibonacci heapa, worst-case analiza složenosti, o kojoj smo do sada govorili nam neće biti dovoljna;
- Nažalost, u ovom slučaju ovakva analiza nije dovoljno dobar pokazatelj stvarne složenosti, pošto su "skupe" operacije relativno retke;
- Zato uvodimo pojam **amortizovane analize**...

Amortizovana analiza

- Neformalno, u amortizovanoj analizi smatramo da se cena povremenih, skupih operacija može u nekim slučajevima "raspodeliti" među čestim, jeftinim operacijama, u svrhu obračunavanja cene;
- Amortizovana analiza **i dalje** daje gornju granicu složenosti (tj. nije probabilistička procena), ali ta granica se odnosi isključivo na niz operacija tokom celog životnog veka strukture podataka;
- Neka operacije zasebno možda prekorači ovako izračunato ograničenje, ali amortizovana granica će i dalje važiti za celu sekvencu.
- Osvrnimo se sada na jedan od metoda amortizovane analize...

Metoda potencijala - neformalan pristup

- Ova metoda se zasniva na intuitivnoj ideji da, kako bismo kompenzovali za povremene, skupe prolaske neke operacije, možemo da "uložimo" neki "novac" negde u strukturi, u svrhu "otplaćivanja" visoke cene te operacije kada ona dođe na red;
- Taj "novac" je u potpunosti fiktivan i **ne** koristi se u implementaciji, služi isključivo u svrhu analize;
- Sada malo formalnije...

Metoda potencijala - formalan pristup

Analizirajmo sekvencu od n uzastopnih operacija nad nekom strukturom. Definišimo sledeće vrednosti za i-tu od tih operacija (gde je $i \in 0, 1, \ldots n$):

- c_i - prava cena operacije
- $\hat{c}_i$ - amortizovana cena operacije
- Φ_i - potencijal strukture sačuvan do i-te operacije

Tada važe sledeće veze:

$$\hat{c}_i = c_i + \Delta\Phi_i = c_i + \Phi_{i+1} - \Phi_i$$

Dakle za sekvencu operacija za i od 0 do $n-1$, ukupna amortizovana cena je:

$$\sum_{i=0}^{n-1} \hat{c}_i = \sum_{i=0}^{n-1} (c_i + \Delta\Phi_i) = \sum_{i=0}^{n-1} c_i + \Phi_n - \Phi_0$$

Metod potencijala - formalan pristup, cont'd

Da bi smo mogli da garantujemo da je ukupna amortizovana cena gornja granica prave cene, mora da važi sledeće:

$$\sum_{i=0}^{n-1} \hat{c}_i \geq \sum_{i=0}^{n-1} c_i$$

Odnosno, oduzimanjem od prethodne jednačine dobijamo:

$$\Phi_n \geq \Phi_0$$

Drugim rečima, ukupan potencijal strukture podataka nikada ne sme pasti ispod svoje početne vrednosti! Prethodni izraz možemo uprostiti tako što proizvoljno postavimo $\Phi_0 = 0$, kada na početku napravimo praznu strukturu, i kao uslov postavimo da potencijal mora u svakom trenutku biti nenegativan.

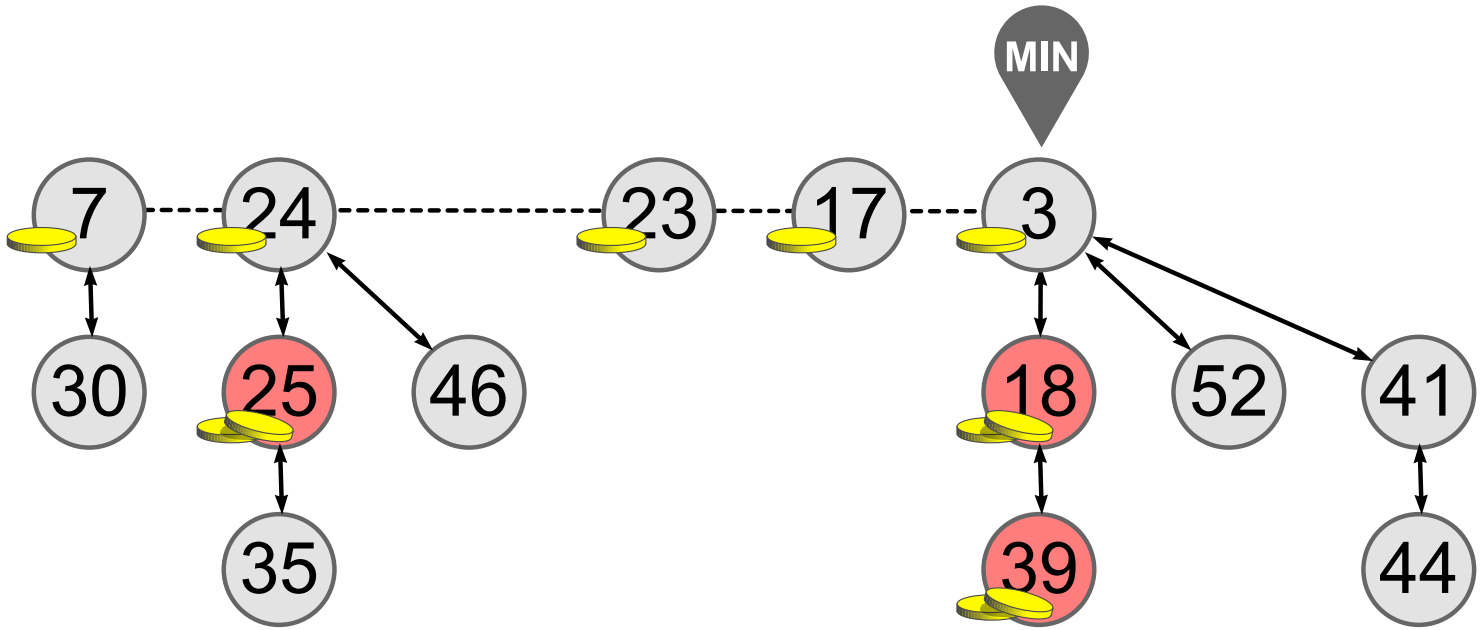
Primena metode potencijala: Fibonacci heap

Ulagaćemo novac na sledeći način:

- **Jedan** novčić: kada dodajemo novi element u listu korenja;
- **Dva** novčića: kada markiramo neki element;

NB Trošimo **jedan** novčić za svaku "osnovnu" operaciju nad heapom (pomeranje/dodavanje/markiranje/uklanjanje elemenata).

Primer ulaganja



Dokaz složenosti: decreasekey()

Prođimo sada ukratko kroz korake metode decreasekey():

- Ukoliko je novi ključ veći od ključa roditelja, trošimo jedan novčić i završili smo (dakle konstantna složenost);
- Primetimo da je isecanje i dodavanje u korenu listu proizvoljnog **markiranog** čvora besplatno! (zašto?)
- Dakle kaskadni rezovi nas ne koštaju ništa, pa jedini doprinos ukupnoj ceni ima prvo sečenje.

NB Konačna složenost je stoga: $O(1)^*$ amortizovano.

Korisni linkovi

Dokaz složenost za extractMin():

`https://en.wikipedia.org/wiki/Fibonacci_heap#Proof_of_degree_bounds`

Kodovi svih heapova koje smo pokrili:

`https://github.com/PetarV-/Algorithms/tree/master/Data%20Structures`

Pregled složenosti rešenja

Operacija	Niz	Binarni	Binomni	Fibonacci
Pravljenje praznog pq-a	$O(1)$	$O(1)$	$O(1)$	$O(1)$
first()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert()	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(1)$
extractMin()	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))^*$
decreaseKey()	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(1)^*$
delete()	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))^*$
merge()	$O(n)$	$O(n)$	$O(\log(n))$	$O(1)$

*amortizovano