III paskaita

Medžio rekursinis metodas

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ – simetrinis

$T(n) = T\left(\frac{n}{a}\right) + T\left(\frac{n}{b}\right) + f(n)$ – nesimetrinis

Pakeitimo metodas

Turime

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

Patikrinsime ar

$$O(n \log_2 n)$$

yra sprendinys.

Pagal apibrėžimą

$$T(n) \le cn \log_2 n$$

Tada

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \le c \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left\lfloor \frac{n}{2} \right\rfloor$$

Sustatome į pradinę lygtį

$$T(n) \le 2\left(c\left\lfloor\frac{n}{2}\right\rfloor \log_2 \left\lfloor\frac{n}{2}\right\rfloor\right) + n \le cn\log_2\frac{n}{2} + n$$
$$= cn\log_2 n - cn\log_2 2 + n = cn\,log_2\, n - cn + n$$
$$\le cn\,log_2\, n$$

Kai $c \ge 1$.

Tikimybinė analizė ir randomizuoti algoritmai

Darbuotojo samdymo uždavinys

Reikia į užimtą darbo vietą pasamdyti geresnį darbuotoją. Darbdaviui nepavykus surasti geresnio darbuotojo pačiam jis sudarė sutartį su įdarbinimo įmone, kad jam kasdieną pokalbiui atsiųstų po vieną pretendentą. Jie sutarė, kad už kiekvieną pretendentą mokės tam tikrą pinigų sumą. Darbdavys nustatęs, kad naujas darbuotojas geresnis, t. y. kvalifikacija aukštesnė už esamą, jį įdarbina. Šis veiksmas darbdaviui atsieina brangiau, nes reikia atleisti einantį pareigas darbuotoją. Darbdavys sutinka su visomis išlaidomis, bet nori įvertinti kaštus.

```
HIRE_ASSISTANT(n)
    Best ← 0
♦ Kandidatas su numeriu 0 – blogiausios kvalifikacijos
♦ fiktyvus kandidatas
    for i ← 0 to n
        do Pokalbis su kandidatu i
            if kandidatas i geresnis už Best kandidatą
                then Best ← i
                    Samdome i kandidatą
```

Pažymėkime c_i – pokalbio kaina, c_h – samdymo kaina. Tegul m – pasamdytų darbuotojų. Tada išlaidos naudojant HIRE_ASSISTANT algoritmą būtų $O(nc_i + mc_h)$. Kiek bebūtų pasamdyta darbuotojų pokalbių reikės pravesti n kartų, todėl duoklė įdarbinimo agentūrai fiksuota ir lygi nc_i.Todėl darbdavys nori minimizuoti išlaidas samdant darbuotojus.

Blogiausias atvejis, kai po kiekvieno pokalbio darbuotojo kvalifikacija pasirodo didesnė ir vis reikia samdyti.

Indikatorinis atsitiktinis dydis

$$\mathrm{I}\{A\} = \begin{cases} 1, & jei\ įvykis\ A\ įvyko \\ 0, & jei\ įvykis\ A\ ne\ įvyko \end{cases}$$

Lema 5.1. Jei S įvykių erdvė, o A priklauso šiai įvykių erdvei. Tarkim $X_A = \mathrm{I}\{A\}$, tada $E[X_A] = P\{A\}$.

Tarkime, kad $X = \sum_{i=1}^{n} X_i$, atsitiktinių indikatorinių funkcijų $X_i = \mathrm{I}\{A | i\, bandymu\}$ suma, tada $E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$, nes dviejų atsitiktinių dydžių sumos vidurkis yra lygus vidurkių sumai.

Lema 5.2. Vykdant pokalbį su kandidatais atsitiktine tvarka, pilna samdymo kaina naudojant HIRE_ASSISTANT algoritmą lygi $O(c_h \ln n)$.

Įrodymas seka iš tokių samprotavimų. Tegul $X = \sum_{i=1}^{n} X_i$ atsitiktinis dydis rodantis kiek reikėjo kartų persamdyti darbuotoją, čia

$$X_i = \begin{cases} 1, & jei\ i-tasis\ kandidatas\ pasamdytas; \\ 0, & jei\ i-tasis\ kandidatas\ nepasamdytas. \end{cases}$$

5.1 lema rodo, kad $E[X_i] = P\{i-tasis\ kandidatas\ pasamdytas\}$

Kadangi darome prielaidą, kad visi kandidatai ateina pokalbiui atsitiktine tvarka. Tada prieš i-tąjį kandidatą, bet kuris kandidatas gali būti geriausias su vienoda tikimybe. Taigi, tikimybė, kad i-tojo pretendento kvalifikacija bus didžiausia yra lygi $\frac{1}{i}$. Todėl, $E[X_i] = \frac{1}{i}$.

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1).$$

Vieno persamdymo kaštai c_h, todėl vidutiniai kaštai $O(c_h \ln n)$.

Norint išvengti nesąžiningo įdarbinimo agentūros veiksmų, reikia darbdaviui pačiam suformuoti atsitiktinę kandidatų eilę pokalbiui. Šiuo atveju algoritmas nedaug skiriasi nuo ankstesniojo.

```
RANDOMIZED_HIRE_ASSISTANT(n)
    Duomenų randomizacija
    Best ← 0
♦ Kandidatas su numeriu 0 – blogiausios kvalifikacijos
♦ fiktyvus kandidatas
    for i ← 0 to n
        do Pokalbis su kandidatu i
            if kandidatas i geresnis už Best kandidatą
                then Best ← i
                    Samdome i kandidatą
```

Vienas iš variantų atsitiktinio pertvarkymo yra algoritmas:

```
PERMUTE_BY_SORTING(A)
    n ← length[A]
    for i ← 1 to n
        do P[i]=RANDOM(1,n^3)
    Surikiuoti masyvą A su
    rikiavimo raktu P
    return A
```

Pavyzdys:

Turime masyvą $A = \langle 1,2,3,4 \rangle$, sugeneruojame masyvą $P = \langle 36,3,97,19 \rangle$. $1 \mapsto 36$, $2 \mapsto 3$, $3 \mapsto 97$, $4 \mapsto 19$ todėl rezultatas bus $\langle 2,4,1,3 \rangle$, nes surikiavus prioritetų masyvą gauname $\langle 3,19,36,97 \rangle$.

Prioritetų generavimas nuo $1..n^3$ sumažina tikimybę, kad pasikartos du vienodi prioritetai.

Lema 5.4. Tarkime, kad visi prioritetai yra skirtingi. Vykdant PERMUTE_BY_SORTING procedūrą gaunama tolygiai pasiskirstęs atsitiktinis perstatymas.

Kadangi vykdomas rikiavimas PERMUTE_BY_SORTING algoritmo sudėtingumas $O(n \log_2 n)$.

Kitas mažiau darbo reikalaujantis algoritmas:

```
PERMUTE_IN_PLACE(A)
    n ← length[A]
    for i ← 1 to n
            do A[i]↔A[RANDOM(i,n)]
    return A
```

Lema 5.5. Vykdant PERMUTE_IN_PLACE procedūrą gaunama tolygiai pasiskirstęs atsitiktinis perstatymas.

PERMUTE_IN_PLACE algoritmo sudėtingumas $O(n)$.

Rikiavimo algoritmai

Pirmoje paskaitoje nagrinėjome rikiavimo algoritmus įterpimo ir suliejimo algoritmus (MERGE_SORT) parodėme, kad pirmojo darbo laikas $T(n) = O(n^2)$, o antrojo – $T(n) = O(n \log_2 n)$.

Nagrinėsime du algoritmus $O(n \log_2 n)$ sudėtingumo:

- Piramidės rikiavimo algoritmas (Heap sort);
- Greito rikiavimo algoritmas (Quick sort);

Abiem atvejais uždavinį formuluosime kaip ir anksčiau:

Rikiavimo uždavinys

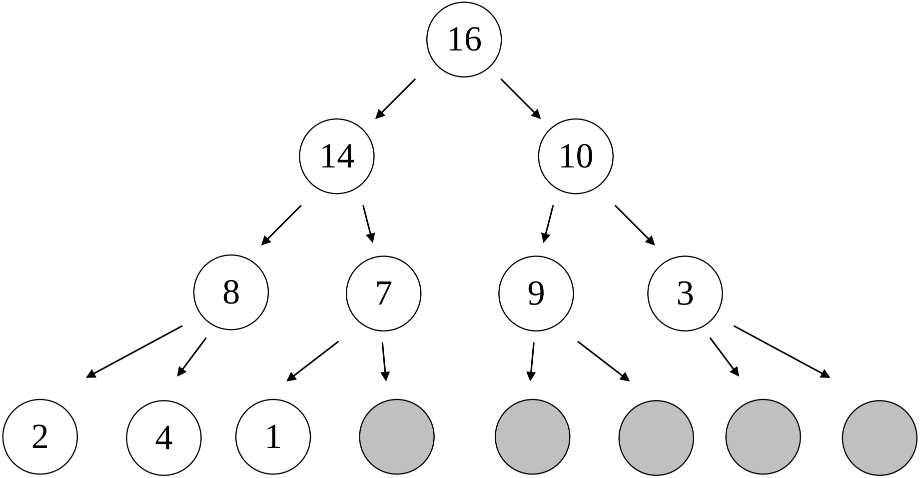
- **Įėjimas:** seka iš *n* skaičių $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Išėjimas:** perstatymas (tvarkos pakeitimas) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ įėjimo sekos, kurios nariai tenkina sąryšį $a'_1 \le a'_2 \le \ldots \le a'_n$.

Piramidės rikiavimo algoritmas

Duomenų stuktūra

Piramidė (binary heap) – duomenų struktūra, t. y. beveik pilnas binarinis medis, kuris išreiškiamas masyvu.

Šio binarinio medžio mazgai atitinka vieną iš masyvo elementų. Visuose medžio lygiuose išskyrus galbūt tik apatinį.


16
14
10
8
7
9
3
2
4
1

Masyvas išreiškiantis piramidę A yra objektas su dviem atributais: $length[A]$, t. y. elementų skaičius masyve ir $heap\_size[A]$, kuris nusako masyvo elementų skaičių piramidėje.

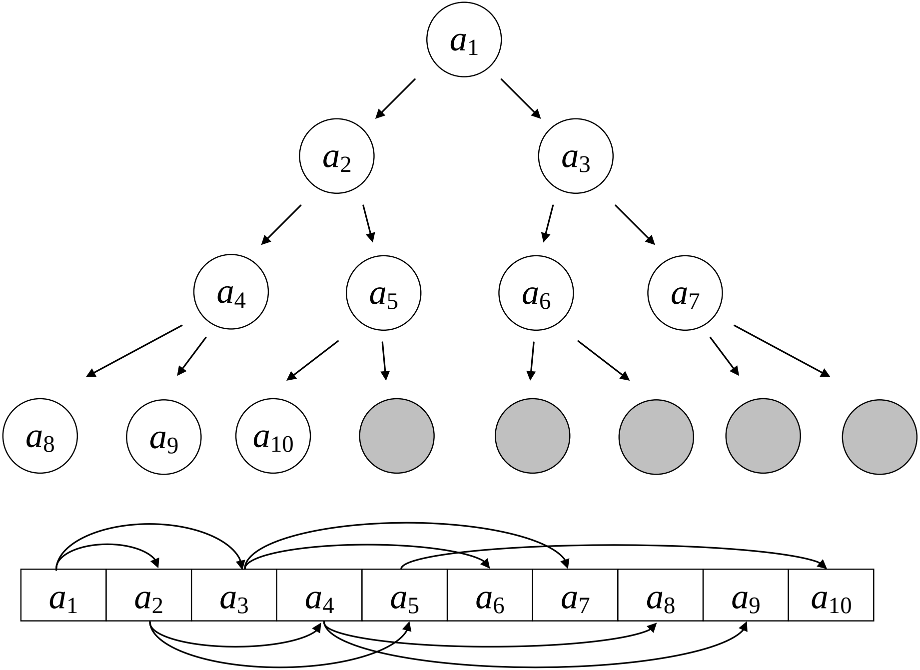
Kitais žodžiais tariant masyve $A[1 \ldots length[A]]$ visi elementai gali būti piramidės elementai ir nė vienas einantis po elemento $A[heap\_size[A]]$, čia $heap\_size[A] \leq length[A]$, nėra piramidės elementai.

Medžio šaknyje randasi $A[1]$ masyvo elementas, toliau jis sudaromas taip: jei kuriam nors medžio mazgui atitinka i-tasis A masyvo elementas, tai jo tėvinis mazgas randamas procedūra PARENT(i), vaikiniai mazgai procedūromis LEFT(i) ir RIGHT(i) .

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

LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$



a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
a_9
a_{10}
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}

Jei piramidės elementai tenkina savybę $A[Parent(i)] \geq A[i]$, tada piramidę vadinsime nedidėjančia. Priešingu atveju, jei $A[Parent(i)] < A[i]$ – piramidė bus nemažėjanti.

Piramidės su n elementų aukštis h – briaunų skaičius pačioje ilgiausioje šakoje:

$$1 + \sum_{i=0}^{h-1} 2^i \leq n \leq \sum_{i=0}^{h} 2^i$$

$$1 + \frac{2^h - 1}{2 - 1} \leq n \leq \frac{2^{h+1} - 1}{2 - 1}$$

$$2^h \leq n \leq 2^{h+1} - 1$$

$$2^h \leq n < 2^{h+1}$$

$$\log_2 2^h \le \log_2 n < \log_2 2^{h+1}$$

$$h \le \log_2 n < h+1$$

$$\log_2 n - 1 < h \le \log_2 n$$

$$c_1 \log_2 n \le \log_2 n - 1 < h \le \log_2 n \le c_2 \log_2 n$$

Kai $n \ge 3$, $c_1 = \frac{1}{2}$, $c_2 = 1$ galioje nelygybė $c_1 \log_2 n \le \log_2 n - 1$, nes

$$\sqrt{n} \le n - 1$$

Tokiu atveju parodėme, kad piramidės aukštis $h = \Theta(\log_2 n)$.

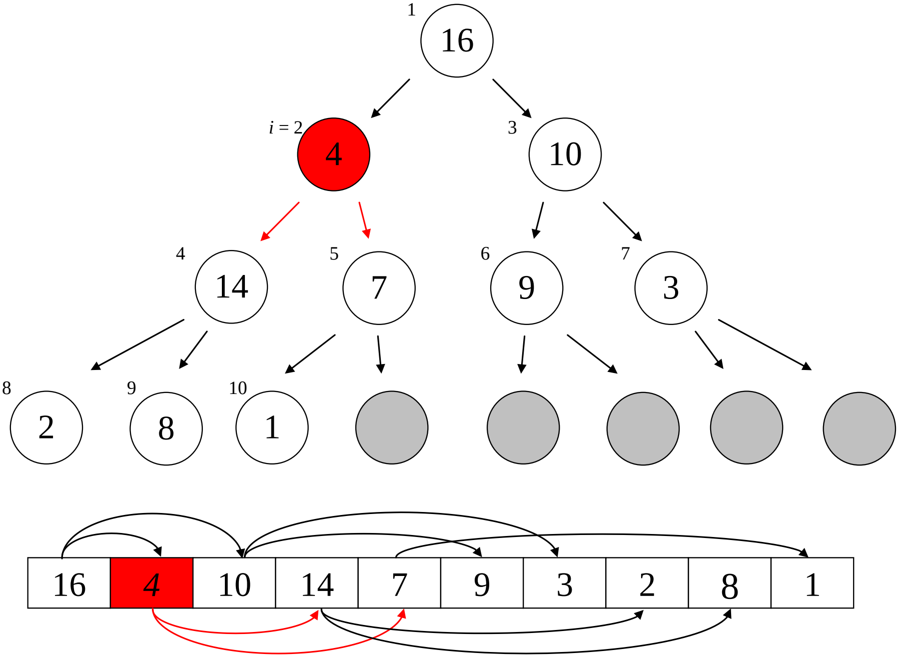
Rikiavimo algoritmas atliekamas tokiu būdu:

- Sudaroma nemažėjanti arba nedidėjanti piramidė
- Piramidės viršūnės elementas yra paskutinis rikiuojamo masyvo dalies elementas, todėl sukeičiami pirmas masyvo elementas su paskutiniu piramidės elementu vietomis.
- Sumažinamas piramidės dydis vienetu ir atliekamas sutvarkymas, kad ji vėl taptų nemažėjančia arba nedidėjančia piramide.
- paskutiniai etapai kartojami tol kol piramidę sudaro tik vienas elementas.

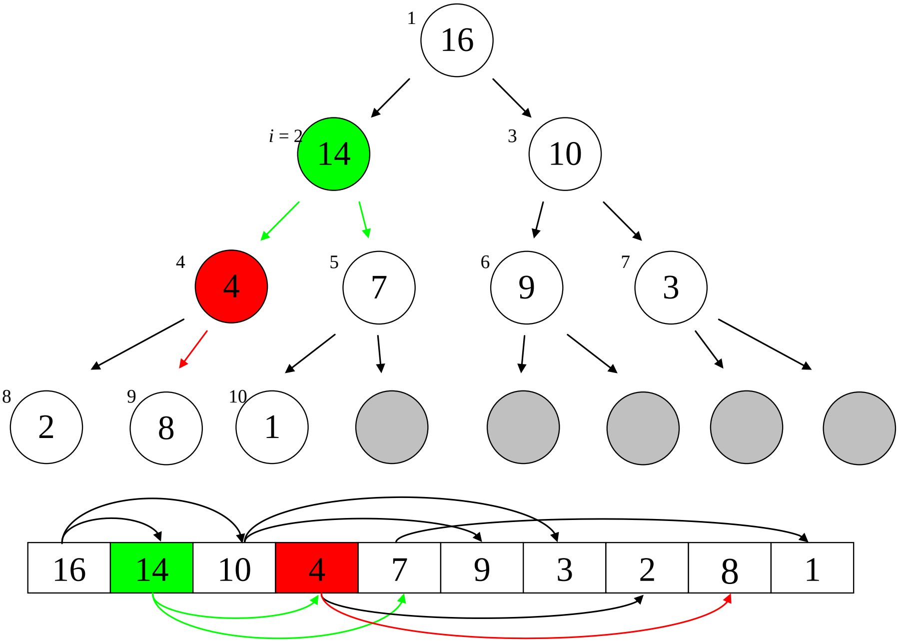
Pagalbinė procedūra netvarkai piramidėje sutvarkyti. Daroma prielaida, kad binariniai medžiai, kurių šaknys yra mazgai LEFT(i) ir RIGHT(i) sudaro nemažėjančias piramides, o mazgas i gali ir netenkinti šios sąlygos. Žemiau esanti procedūra nuleidžia i-tąjį mazgą žemyn, kad ši sąlyga būtų tenkinama ir i-tam mazgui.

MAX_HEAPIFY(A, i)

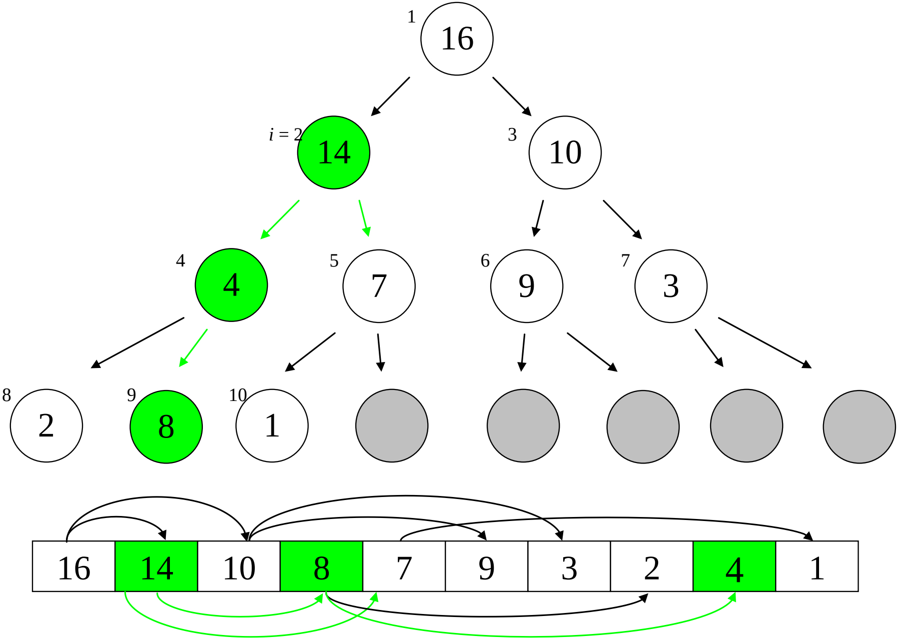
```
l ← Left(i)
r ← Right(i)
if l ≤ heap_size[A] ir A[l] > A[i]
 then largest ← l
      else largest ← i
if r ≤ heap_size[A] ir A[r] > A[largest]
 then largest ← r
if largest ≠ i
 then A[i] ↔ A[largest]
      MAX_HEAPIFY(A, largest)
```



1
16
i = 2
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
16 4 10 14 7 9 3 2 8 1



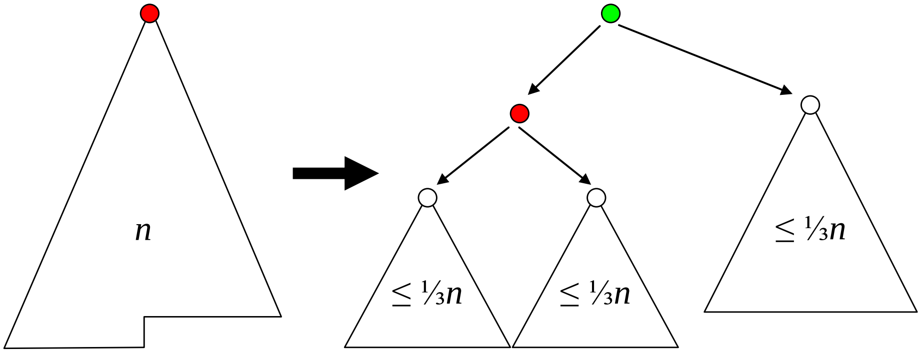
1
16
i = 2
14
3
10
4
4
5
7
6
9
7
3
8
2
9
8
10
1
16
14
10
4
7
9
3
2
8
1



1
16
i = 2
14
3
10
4
4
5
7
6
9
7
3
8
2
9
8
10
1
16
14
10
8
7
9
3
2
4
1

Sudėtingumas randamas iš rekurentinės nelygybės $T(n) \le T\left(\frac{2n}{3}\right) + \Theta(1)$,

nes 1–9 eilutės atliekamos per konstantą, o 10 eilutė nagrinėja mažiau nei $\frac{2n}{3}$ elementų.



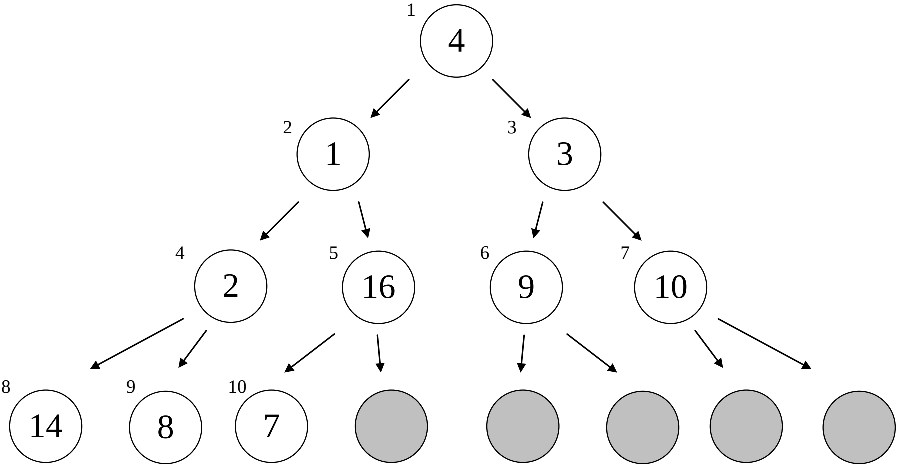
Remiantis pagrindinės teoremos 2 dalimi sprendinys yra lygus $T(n) = O(\log_2 n)$. Arba galima išreikšti per piramidės aukštį $T(n) = O(h)$.

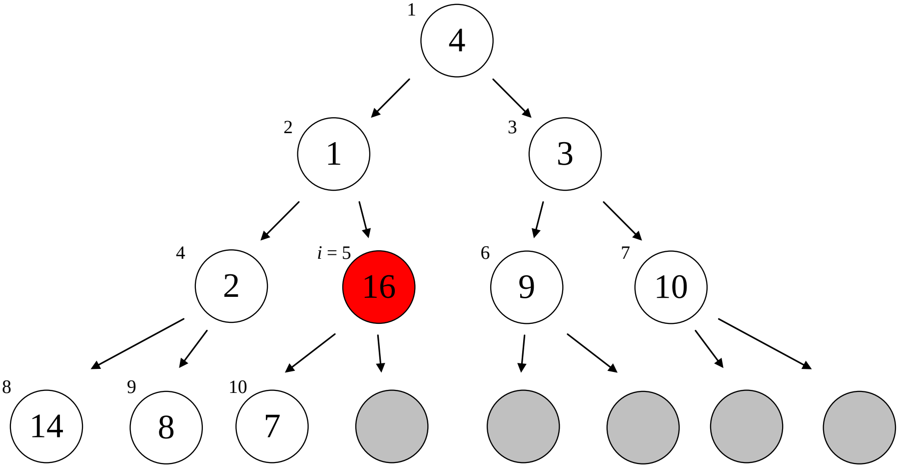
Piramidės sudarymo procedūra

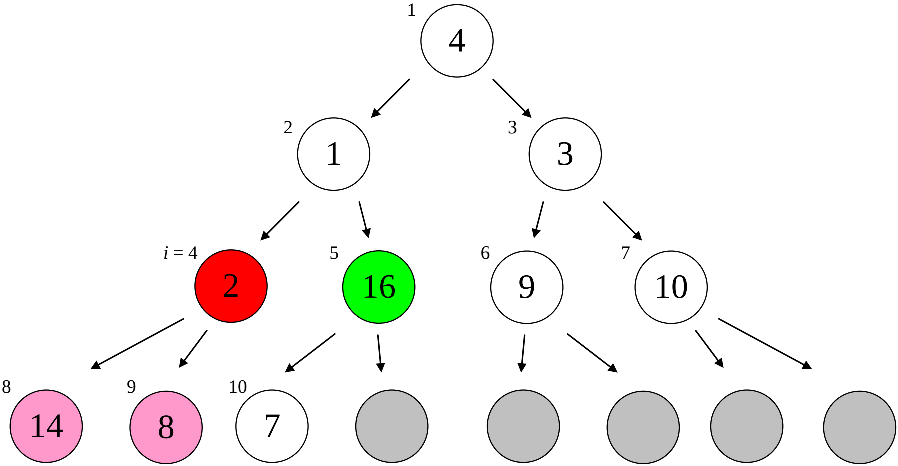
BUILD_MAX_HEAP(A)

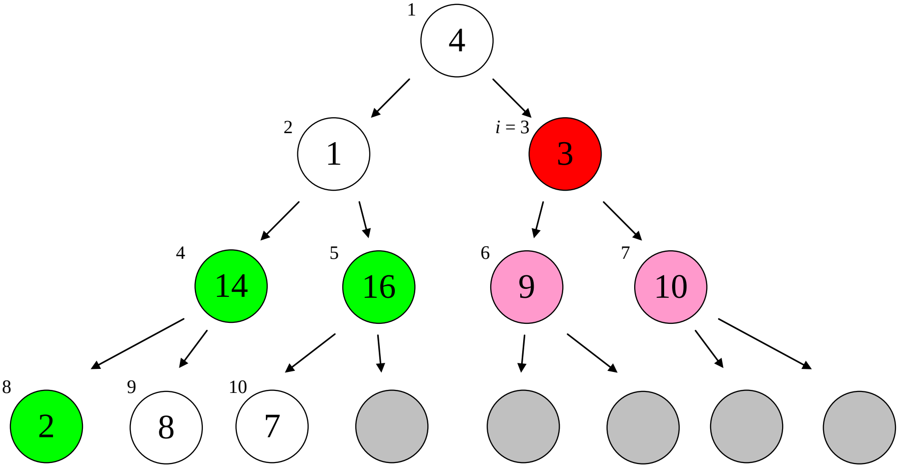
1. *heap_size[A]* ← *lenght[A]*
2. ***for*** *i* ← $\lfloor length[A]/2 \rfloor$ ***downto*** 1
3. ***do*** *MAX_HEAPIFY(A, i)*

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

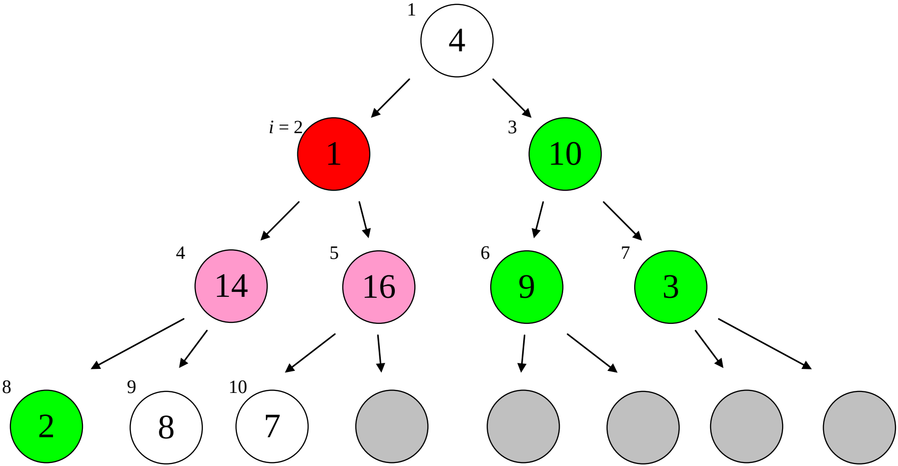


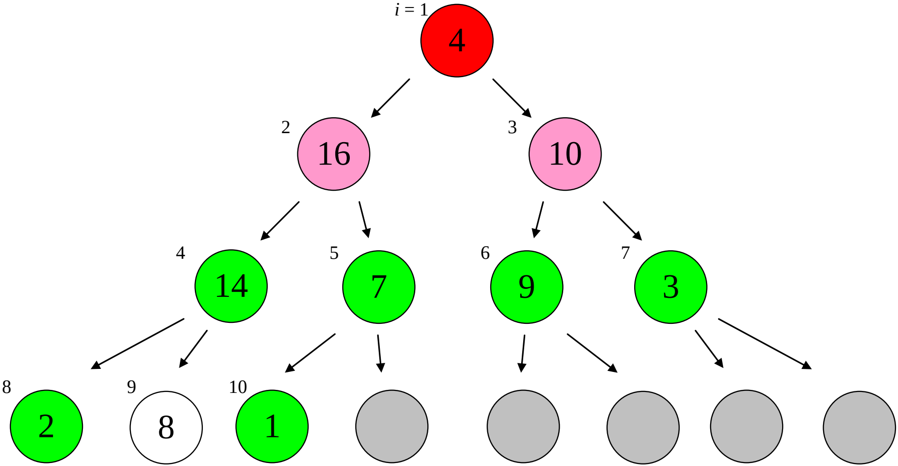

1
4
2
1
3
3
4
2
i = 5
16
6
9
7
10
8
14
9
8
10
7


1
4
2
1
3
3
i = 4
2
5
16
6
9
7
10
8
14
9
8
10
7

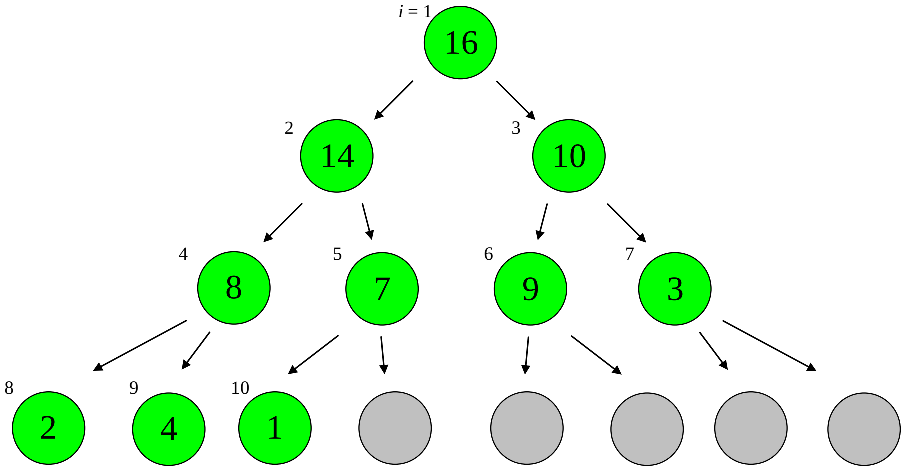


1
4
2
1
i = 3
3
4
14
5
16
6
9
7
10
8
2
9
8
10
7


1
4
i = 2
1
3
10
4
14
5
16
6
9
7
3
8
2
9
8
10
7



i = 1
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1



i = 1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1

Procedūros korektiškumo įrodymas

Suformuluojame ciklo invariantą: prieš kiekvieną ciklo (2, 3 eilutės) visi mazgai su indeksais $i+1,\ i+2,\ \ldots,\ n$ yra šaknys nedidėjančios piramidės.

INICIALIZACIJA: pradedant ciklą $i = \lfloor n/2 \rfloor$, o mazgai su indeksais $\lfloor n/2 \rfloor + 1,\ \lfloor n/2 \rfloor + 2, \ldots,\ n$ yra medžio lapai, todėl kiekvienas iš jų yra triviali nedidėjanti piramidė.

IŠLIEKAMUMAS: Prieš įsitikinat, kad kiekviena iteracija išsaugo invariantą, pastebėsime, kad vaikiniai mazgai i mazgo turi numerius didesnius nei i. Remiantis ciklo invariantu jie yra nedidėjančios piramidės. Tai kaip tik ta sąlyga kada iškviečiama procedūra MAX_HEAPIFY(A, i), kad pertvarkyti mazgą į nemažėjančią piramidę. Be to, iškvietus procedūrą MAX_HEAPIFY(A, i) išsaugoma piramidės savybė visiems mazgams su indeksais $i+1$, $i+2$, …, n, kurie nemažėjančios piramidės.

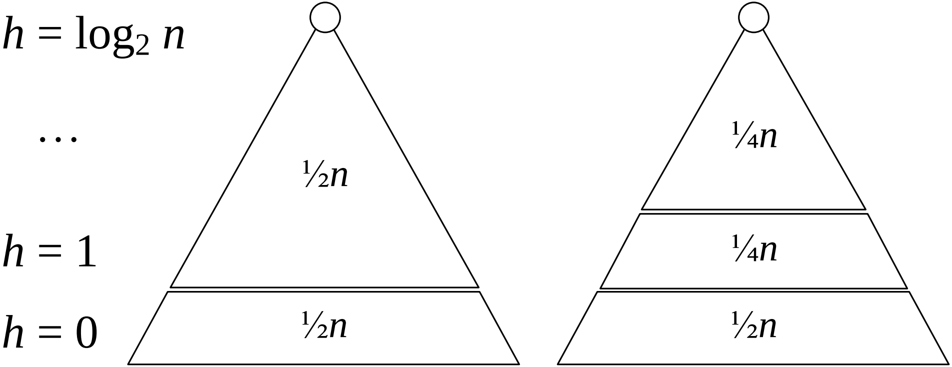
PABAIGA: Baigus ciklą $i=0$. Todėl remiantis invariantu visi mazgai su indeksais 1, 2, …, n yra nemažėjančios piramidės. Mūsų atveju mazgas su numeriu 1 ir yra sudaroma ne mažėjanti piramidė.

Sudėtingumo įvertinimas

Grubus įvertinimas procedūros BUILD_MAX_HEAP(A) – $O(n\log_2 n)$, nes vykdomas ciklas $\lfloor n/2 \rfloor$ kartų, o procedūros MAX_HEAPIFY(A, i) darbo laikas – $O(\log_2 n)$.

Nors šis įvertis yra visiškai korektiškas, bet nėra asimptotiškai tikslus. Tai seka iš fakto, kad procedūros darbo laikas priklauso nuo i-tojo mazgo aukščio h, todėl procedūros MAX_HEAPIFY(A, i) darbo laikas – $O(h)$.

Pagalbinis faktas, kad kiekviename lygyje yra $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ mazgų (6.3-3 pratimas). Parodysime, pilno binarinio medžio atveju.

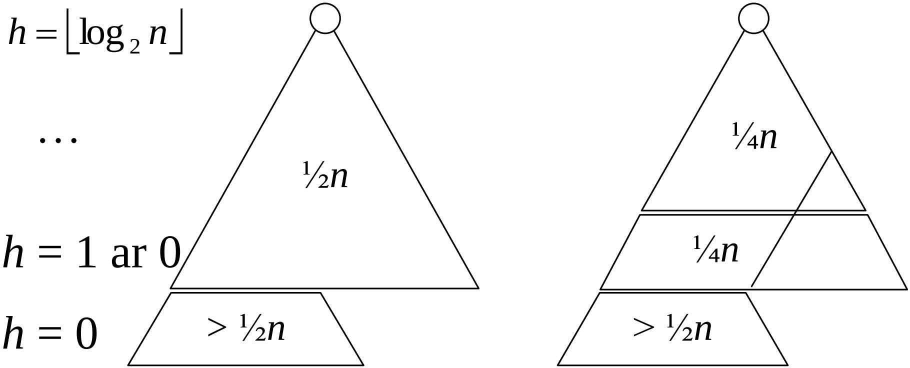


$$n = \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \frac{n}{2^4} + \ldots + \underbrace{\frac{n}{2^{h+1}}}_{1}$$

$$1 = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \ldots + \frac{1}{2^{h+1}}$$

$$1 = \frac{\left(\frac{1}{2}\right)^{h+1} - 1}{\frac{1}{2} - 1} = 2 - \left(\frac{1}{2}\right)^{h}$$

$$\underline{h = \log_2 n}$$



Šiuo atveju BUILD_MAX_HEAP(A) darbo laikas išreiškiamas tokiu būdu:

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

Žinoma, kad $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, kai $|x| < 1$.

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

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

Kodėl galioje ši lygybė? O reiškia konstantos tikslumu.

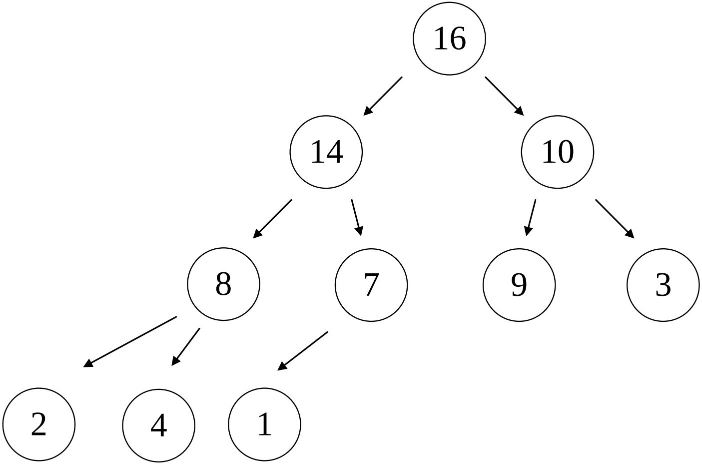
$$O\underbrace{\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^{h+1}} \right)}_{<2n} = O\underbrace{\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} \right)}_{=2n} = O\underbrace{(n)}_{=1n}$$

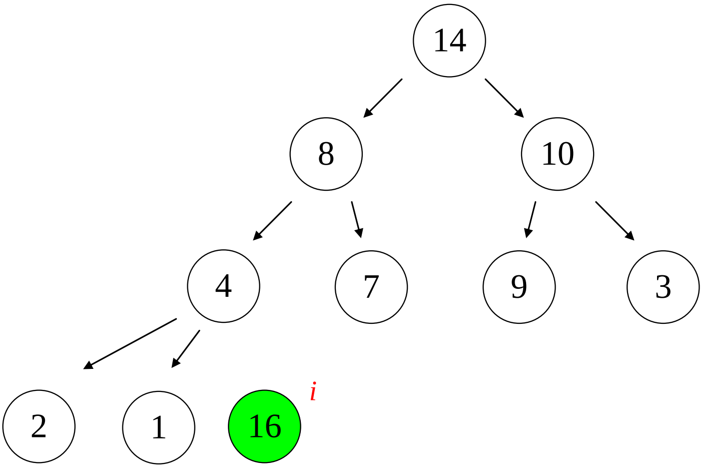
Taigi procedūros BUILD_MAX_HEAP(A) darbo laikas – $O(n)$.

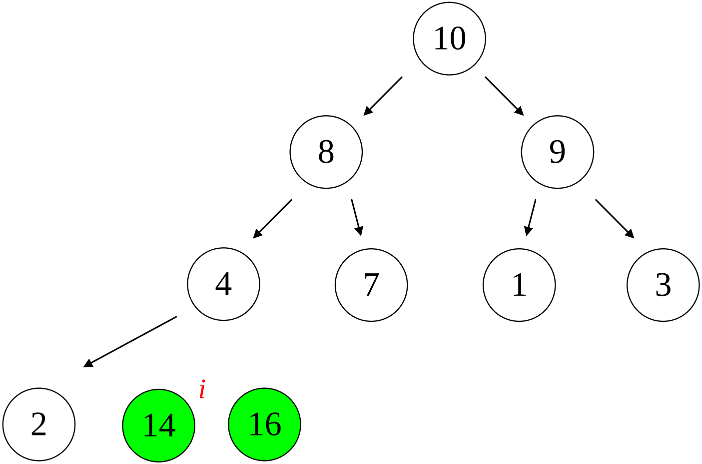
Rikiavimas piramide algoritmas

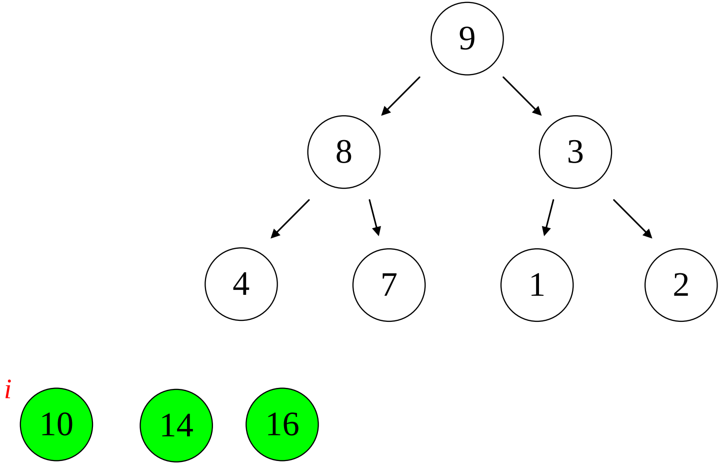
```
HEAPSORT(A)
1. BUILD_MAX_HEAP(A)
2. heap_size[A] ← lenght[A]
3. for i ← lenght[A] downto 2
4.  do A[1] ↔ A[i]
5.        heap_size[A] ← heap_size[A]
6.        MAX_HEAPIFY(A, 1)
```

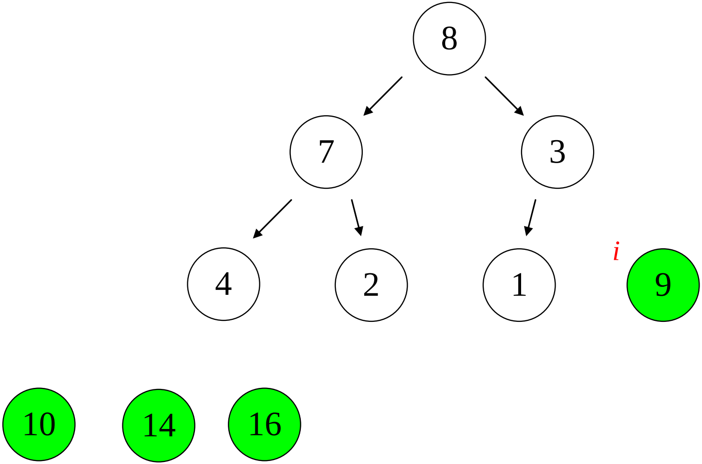
Kaip dirba algoritmas?

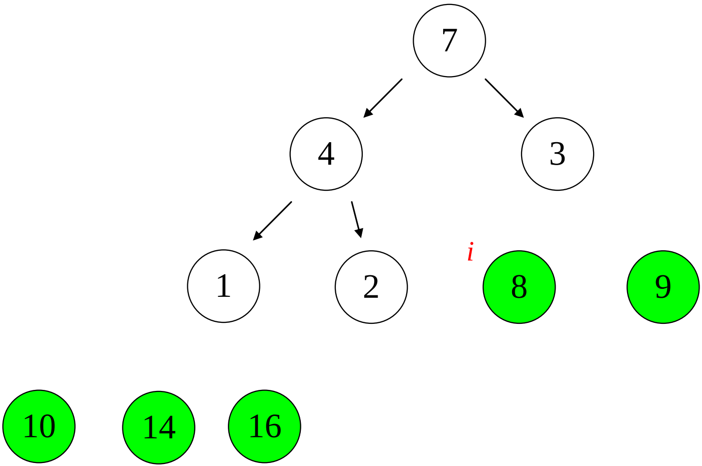

16
14
10
8
7
9
3
2
4
1

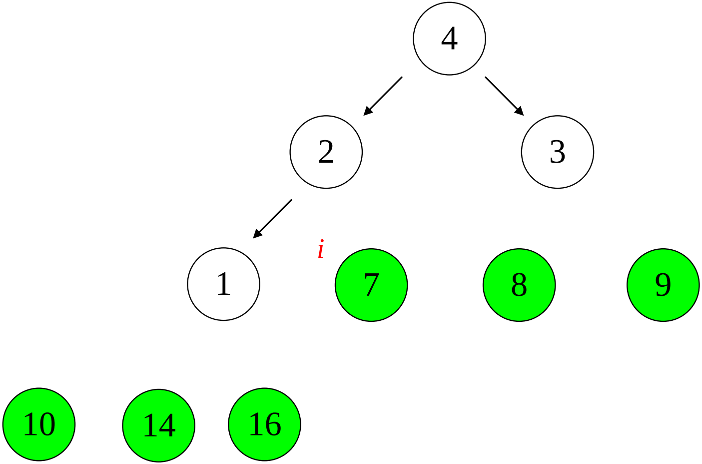

14
8
10
4
7
9
3
2
1
16
i

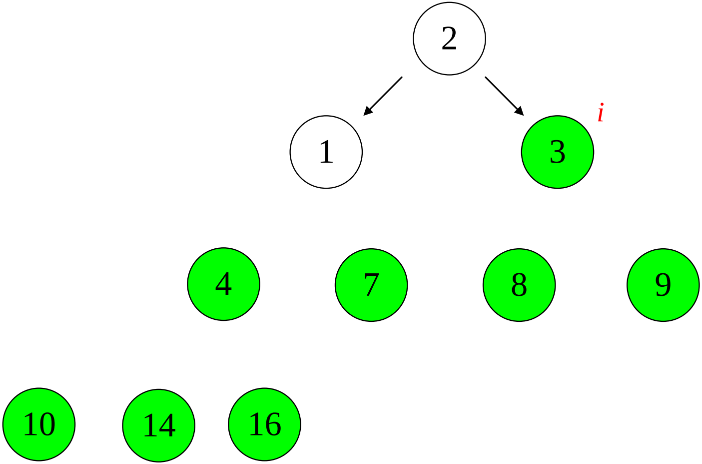

10
8
9
4
7
1
3
2
14
i
16

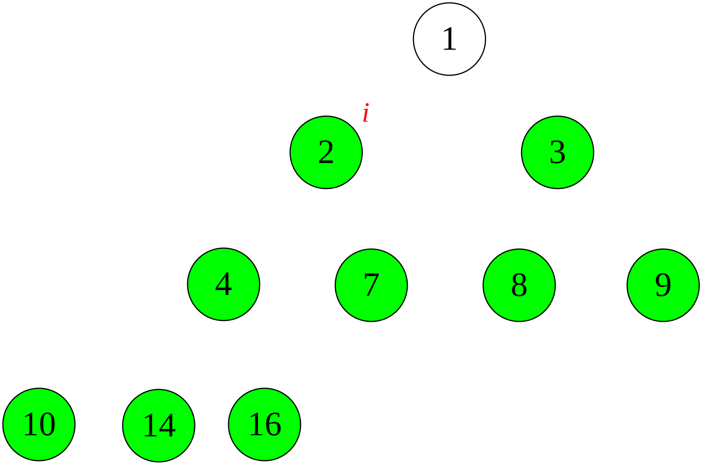

9
8
3
4
7
1
2
i
10
14
16


8
7
3
4
2
1
i
9
10
14
16


7
4
3
1
2
i
8
9
10
14
16


4
2
3
1
i
7
8
9
10
14
16


2
1
3
i
4
7
8
9
10
14
16



Rezultatas

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

Korektiškumo įrodymas

Savarankiškai.

Sudėtingumo įvertinimas

Procedūros darbo laikas $O(n\log_2 n)$, nes $n-1$ kartą kviečiama procedūra MAX_HEAPIFY(A, 1), kurios darbo laikas $O(\log_2 n)$, o procedūros BUILD_MAX_HEAP(A) darbo laikas – $O(n)$.

$(n-1)O(\log_2 n)+O(n)=O(n\log_2 n)+O(n)=O(n\log_2 n)$, nes $(n-1)c_1\log_2 n\le c_2\log_2 n$, kai $c_1=c_2$ visiems $n>2$.

Eilės su prioritetais

Eilė su prioritetais – tai duomenų struktūra skirta aptarnauti S aibę, su kurios kiekvienu elementu siejamas raktas key.

Eile su nedidėjančiais prioritetais (raktais) palaiko šias operacijas:
INSERT(S, x) – įterpimas elemento x į aibę S.
MAXIMUM(S) – grąžina didžiausią elementą iš aibės S.
EXTRACT_MAX(S) – grąžina didžiausią elementą iš aibės S ir pašalina.
INCREASE_KEY(S, x, k) – padidina aibės S elemento x raktą raktu k.

Eile su nemažėjančiais prioritetais (raktais) palaiko šias operacijas:
INSERT(S, x) – įterpimas elemento x į aibę S.
MINIMUM(S) – grąžina mažiausią elementą iš aibės S.
EXTRACT_MIN(S) – grąžina didžiausią elementą iš aibės S ir pašalina.
DECREASE_KEY(S, x, k) – sumažina aibės S elemento x raktą raktu k.

Šiuo atveju yra masyvas A ir yra aibė S.

HEAP_MAXIMUM(A)

1. ***return*** *A[1]*

Sudėtingumas – $\Theta(1)$.

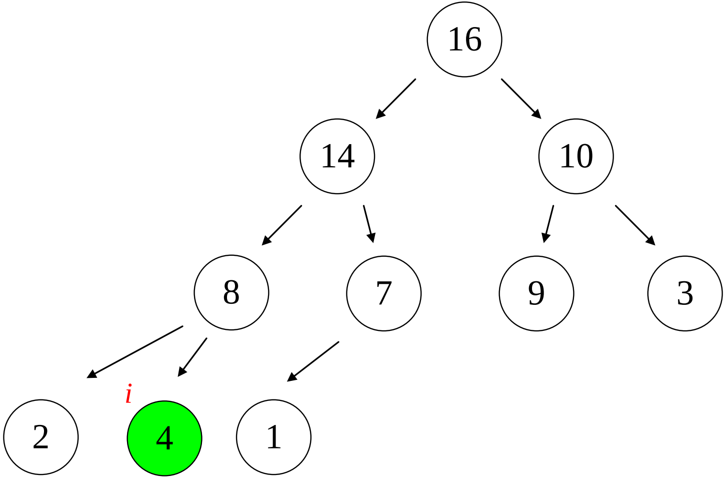
HEAP_EXTRACT_MAX(A)

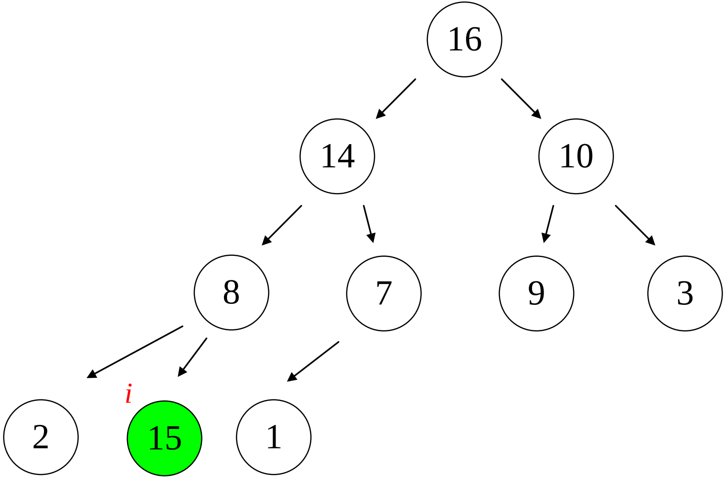
1. ***if*** *heap_size[A]*< 1
2. ***then error*** *„Eilė tuščia"*
3. *max* ← *A[1]*
4. *A[1]* ← *A[heap_size[A]]*
5. *heap_size[A]* ← *heap_size[A]*- 1
6. *MAX_HEAPIFY(A, 1)*
7. ***return*** *A[1]*

Sudėtingumas – $O(\log_2 n)$.

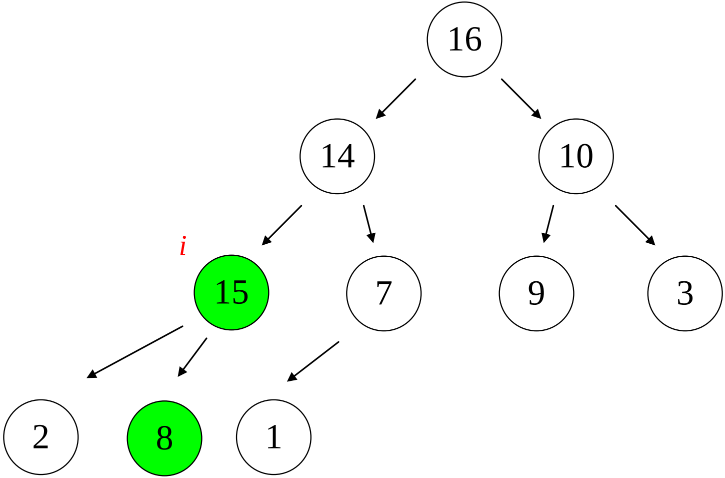
```
HEAP_INCREASE_KEY(A, i, key)
   if key < A[i]
        then error „naujas raktas mažesnis už einamąjį“
   A[i] ↔ key
   while i > 1 ir A[PARENT(i)] < A[i]
     do A[i] ↔ A[PARENT(i)]
          i ← PARENT(i)
```

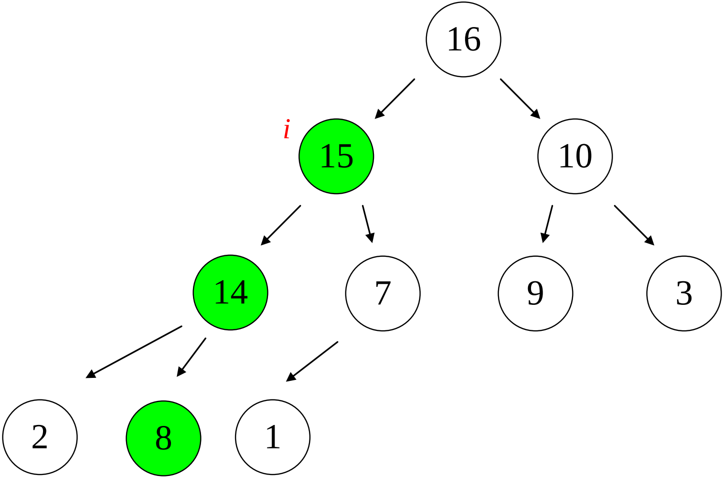
Kaip dirba procedūra?


16
14
10
8
7
9
3
2
i
4
1



16
14
10
8
7
9
3
2
i
15
1


16
14
10
i
15
7
9
3
2
8
1



Sudėtingumas – $O(\log_2 n)$, nes 4-6 eilutės gali būti vykdomos tiek kartų koks yra piramidės aukštis.

MAX_HEAP_INSERT(*A*, *key*)

1. *heap_size[A] ← heap_size[A]+ 1*
2. *A[heap_size[A]] ← -∞*
3. *HEAP_INCREASE_KEY(*A, *heap_size[A]*, key*)*

Sudėtingumas – $O(\log_2 n)$.