# DUOMENŲ STRUKTŪROS IR ALGORITMAI

MARIUS GŽEGOŽEVSKIS

# Algoritmų analizė ir sudėtingumas

**Algoritmų analizė** – labai svarbus darbo etapas, kuriant ir detalizuojant algoritmus, nusakant jų veikimą ir efektyvų taikymą praktiniams uždaviniams. Kadangi ne su kiekviena programa galima atlikti daug bandymų ar nuodugniai matematiškai įvertinti, analizė atliekama algoritmų klasėms.

# Algoritmų analizė ir sudėtingumas

**Algoritmų architektūra ir jų analizė** dažnai labai susiję. Mažų ir nesudėtingų programų algoritmai dažniausiai realizuojami tiesmukiškai, tačiau jei algoritmai yra didelės sistemos dalis, naudojami abstraktūs duomenų tipai arba panaši vykdymo metodika, kuri įgalina patikimai pereiti nuo analizuojamo algoritmo varianto prie jo detalesnės realizacijos.

# Algoritmų analizė ir sudėtingumas

Toliau yra pateiktos, nagrinėjamų algoritmų realizacijos naudojamos didelėse programose, operacinėse ir taikomosiose sistemose.

Todėl svarbu atkreipti dėmesį į jų savybes, didinančias patikimumą. Pavyzdžiui, kuriant realias sistemas geriau naudoti programavimo aplinką, kuri gali skirtis nuo naudojamos, tačiau kuri praneša apie klaidingas sąlygas, įgalina programas lengvai keisti, sieti jas su kitomis dalimis ar komponentais, įgalina jas lengvai perkelti į kitas terpes.

# Analizuojant algoritmus, tikslinga:

- ✓Algoritmą modifikuoti ar pakeisti jį kitu, labai panašiu, tačiau daug lengviau analizuojamu algoritmu;
- ✓parinkti įvedimo duomenis, kurie patikimai atspindėtų įvairius algoritmo darbo atvejus, pvz., tikslinga spręsti, ar naudoti realius ar atsitiktinius, ar nepagrįstus duomenis (realūs duomenys tiksliau įvertina programos efektyvumą; atsitiktiniai duomenys garantuoja, kad algoritmas bus patikrintas eksperimentiškai; nepagrįsti duomenys rodo, kaip algoritmas veikia įvairiais pradinių duomenų atvejais, pvz. kaip veikia rūšiavimo algoritmas kai visi įvedimo duomenys yra lygūs),

# Analizuojant algoritmus, tikslinga:

- ✓įvertinti operacijas atsižvelginat į jų svarba bei abstraktumo lygį ir analizuoti algoritmo veikimą šių operacijų pagrindu.
- ✓Kadangi algoritmui realizuoti naudojamas kompiuteris, algoritmų analizei keliamas uždavinys įvertinti, kaip ilgai truks skaičiavimai. Tačiau laiko faktorius, nors ir dažniausiai naudojamas, yra ne vienintelis galimas. Algoritmo atliekamų 2 operacijų skaičių, t.y. **algoritmo sudėtingumą, galima matuoti,** remiantis įvairiais faktoriais:

# Algoritmų sudėtingumo vertinimo faktoriai:

- ✓ laiku (laikinis sudėtingumas – time complexity, dažniausiai sutinkamas algoritmų analizėje),
- ✓ naudojamos atminties dydžiu (erdvinis sudėtingumas – space complexity),
- ✓ techninės įrangos parametrais,
- ✓ informacijos, kurią generuoja gaunami programos rezultatai, kiekiu (information complexity),
- ✓ enrgijos, reikalingos duotam uždaviniui spręsti, kiekiu, kt.

# Algoritmų sudėtingumo vertinimo faktoriai:

Akivaizdu, kad algoritmo vykdymas ir nuo įvairių kitų faktorių, iš kurių svarbiausias yra įvedimo duomenų kiekis (arba panaši **duomenų charakteristika**). Todėl paprastai visi minėti sudėtingumo aspektai skaičiuojami kaip funkcijos nuo pradinių duomenų parametro(-ų), kuris labiausiai įtakoja algoritmo veiklą.

# Algoritmų sudėtingumo vertinimo faktoriai:

Šis parametras (dažnai žymimas **N**), gali būti tiesiog duomenų kiekis, arba tai gali būti daugianario laipsnis, failo dydis, simbolių skaičius teksto eilutėje, kokia nors kita abstrakti išraiška, priklausomai nuo sprendžiamos užduoties.

Dažniausiai šis parametras būna tiesiog proporcingas apdorojamos informacijos kiekiui. Kai naudojami keli parametrai, analizė gali būti redukuojama iki vieno parametro, arba gali būti imamos tinkamos parametrų kombinacijos.

# Algoritmų sudėtingumo vertinimo faktoriai:

**Praktinis algoritmų analizės tikslas** – išreikšti reikalavimus programos ištekliams priklausomai nuo **N** matematinėmis formulėmis, kurios nusako algoritmo elgesį esant didelėms parametrų reikšmėms, t.y. asimptotiškai.

# Asimptotinė analizė

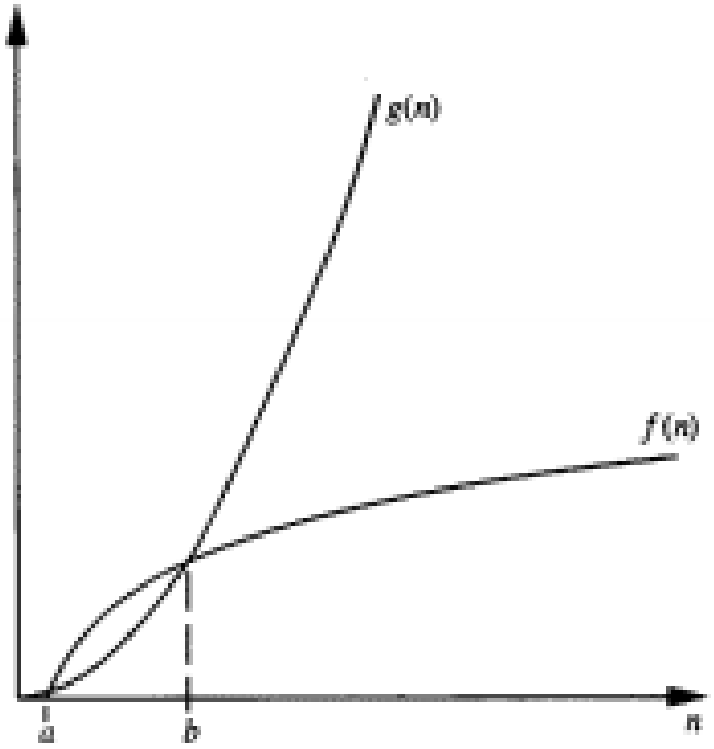
**Asimptotinė analizė** skirta tirti funkcijų (nebūtinai išreikštų matematine kalba) elgesį arba santykį tarp funkcijų, kai jų argumentas(-ai) asimptotiškai artėja prie kokios nors reikšmės(-ių). Taikant analizę galima teigti, kad viena funkcija yra apytikriai (asimptotiškai) lygi kitai funkcijai, arba yra už ją mažesnė (ar didesnė), ar tenkina kokį nors duotą santykį tarp funkcijų.

Populiariausios priemonės, naudojamos tokiems tyrimams, yra **O-žymuo, o-žymuo, Θ-žymuo**, ir pan. Jų apibrėžimuose dalyvauja neneigiamos funkcijos, tačiau tai nesiaurina jų prasmės.

# O-žymuo: Funkcija g(N) yra vadinama O(f(N)), jei egzistuoja konstantos c_0 ir N_0 , tokios , kad  0 < g(N) < c_0,  f (N) visiems N > N_0

Pvz., O(ln x) ir O(x^4), ka

ip parodyta pav. 1.1:




**Pav. 1.1. *ln x vs O* ($x^4$)**

# O-žymuo tinka trims skirtingiems tikslams:

1. siekiant išvengti galimos paklaidos atmetant mažus ar neesminius matematinių formulių narius,
2. siekiant išvengti galimos paklaidos atmetant programos dalis, kurios sudaro visų analizuojamų veiksmų mažą (šalutinę) dalį,
3. klasifikuoti algoritmus pagal viršutinius apribojimus jų bendram vykdymo laikui.

# O-žymuo

Konstantos c _0 ir N_0, dalyvaujančios **O-žymenyje**, paslepia algoritmo realizavimo detales, kurios gali būti svarbios praktikoje. Akivaizdu, kad algoritmo vykdymo laikas O(f(N)) negali teikti informacijos apie vykdymo laiką, jei N yra mažiau nei N_0, o c_0 gali nuslėpti didelį kiekį reikalingų operacijų, siekiant išvengti paties blogiausio atvejo.

Aišku, kad algoritmas, naudojantis **N^2** nanosekundžių, bus pasirinktas vietoj algoritmo, naudojančio **log N** šimtmečių, tačiau tokio pasirinkimo nebus galima padaryti remiantis O-žymeniu.

# O-žymuo

Naudojant O-žymenį, dažnai išraiškas galima užrašyti kompaktiškiau, praleidžiant neesminius narius.

Pavyzdžiui, išplėtus išraišką $(N + O(1))(N + O(\log N) + O(1))$

gaunami šeši nariai:

$$N^2 + O(N) + O(N \log N) + O(\log N) + O(N) + O(1).$$

Tačiau į daugelį narių galima nekreipti dėmesio, paliekant aproksimaciją:

$$N^2 + O(N \log N).$$

# Duomenų struktūrų operacijos

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | - | O(1) | O(1) | O(1) | - | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | - | O(log(n)) | O(log(n)) | O(log(n)) | - | O(n) | O(n) | O(n) | O(n) |
| B-Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | - | O(log(n)) | O(log(n)) | O(log(n)) | - | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |

**LEGENDA -** Excellent | Good | Fair | Bad | Horrible

# MASYVO RŪŠIAVIMO ALGORITMAI

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(log(n)) |
| Mergesort | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(n) |
| Timsort | O(n) | O(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | O(1) |
| Shell Sort | O(n) | O((nlog(n))^2) | O((nlog(n))^2) | O(1) |
| Bucket Sort | O(n+k) | O(n+k) | O(n^2) | O(n) |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n+k) |

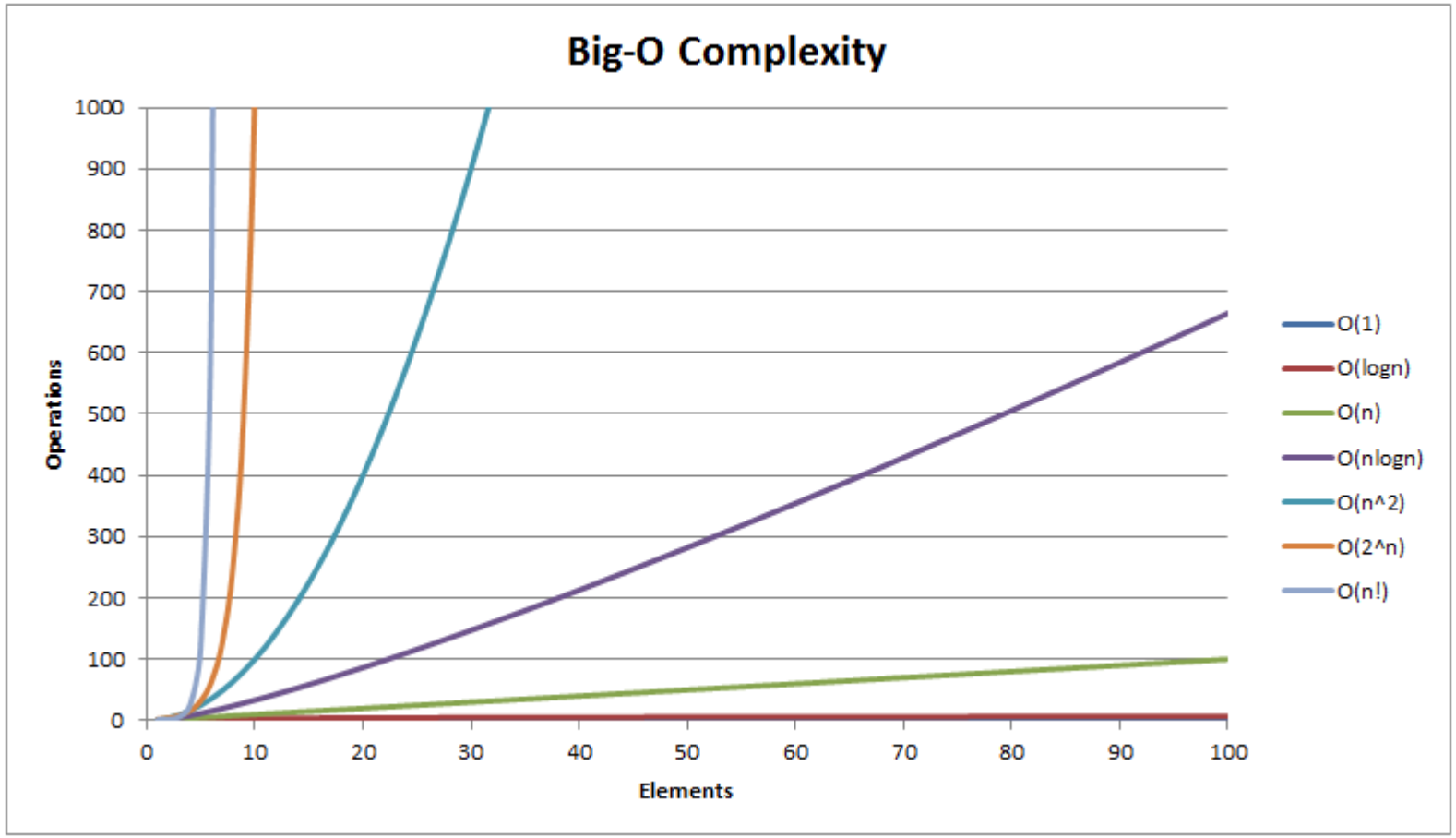
# GRAFO OPERACIJOS

| Node / Edge Management | Storage | Add Vertex | Add Edge | Remove Vertex | Remove Edge | Query |
|---|---|---|---|---|---|---|
| Adjacency list | O(\|V\|+\|E\|) | O(1) | O(1) | O(\|V\| + \|E\|) | O(\|E\|) | O(\|V\|) |
| Incidence list | O(\|V\|+\|E\|) | O(1) | O(1) | O(\|E\|) | O(\|E\|) | O(\|E\|) |
| Adjacency matrix | O(\|V\|^2) | O(\|V\|^2) | O(1) | O(\|V\|^2) | O(1) | O(1) |
| Incidence matrix | O(\|V\| · \|E\|) | O(\|V\| · \|E\|) | O(\|V\| · \|E\|) | O(\|V\| · \|E\|) | O(\|V\| · \|E\|) | O(\|E\|) |

# PIRAMIDĖS OPERACIJOS

| Type | Time Complexity | | | | | | |
|---|---|---|---|---|---|---|---|
| | Heapify | Find Max | Extract Max | Increase Key | Insert | Delete | Merge |
| Linked List (sorted) | - | O(1) | O(1) | O(n) | O(n) | O(1) | O(m+n) |
| Linked List (unsorted) | - | O(n) | O(n) | O(1) | O(1) | O(1) | O(1) |
| Binary Heap | O(n) | O(1) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(m+n) |
| Binomial Heap | - | O(1) | O(log(n)) | O(log(n)) | O(1) | O(log(n)) | O(log(n)) |
| Fibonacci Heap | - | O(1) | O(log(n)) | O(1) | O(1) | O(log(n)) | O(1) |

# Big – O ( Didžiojo O) sudėtingumo lentelė



# Piramidės rūšiavimo algoritmas

Piramidės rūšiavimas naudojamas piramidės duomenų struktūrai t.y. toks dvejetainis medis, kurio šaknis yra didžiausią reikšmę turinti viršūnė.

Piramidės rūšiavimo algoritmas yra sukonstruotas remiantis išrinkimo rūšiavimo algoritmu. Tai lėtesnis algoritmas nei spartusis rūšiavimo algoritmas.

# Kas yra piramidė (angl. Heap) ?

**Piramidė** – tai beveik pilnas dvejetainis medis, kuris tenkina žemiau pateiktą sąlygą: jei B yra viršūnės A vaikas, tai A reikšmė didesnė arba lygi B reikšmei. Tai reiškia, kad viršūnė, turinti didžiausią reikšmę yra šaknis.

# Piramidės pavyzdys

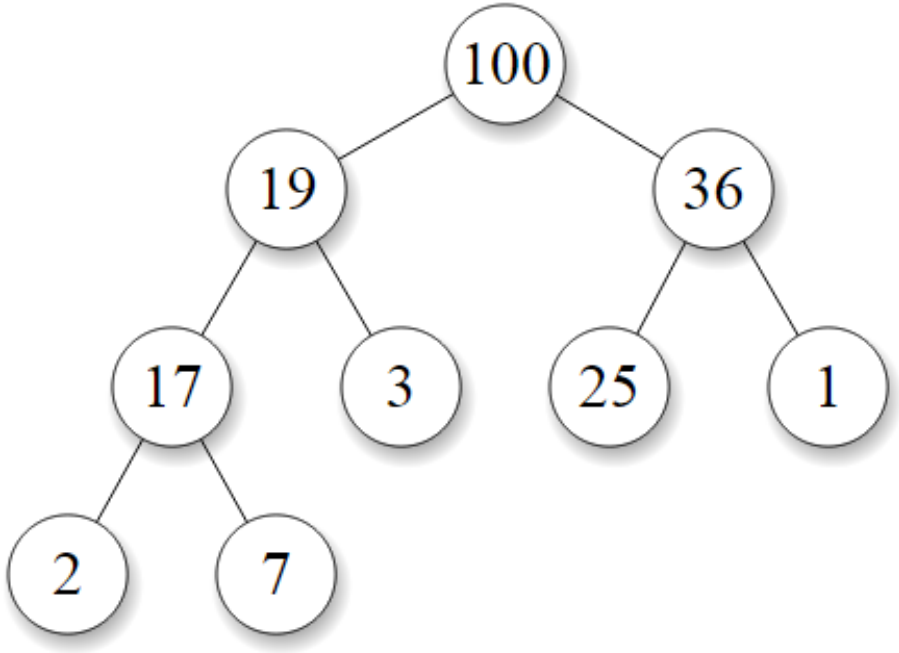
Piramidė konstruojama kaip ir bet kuris pilnas dvejetainis medis t. y. iš kairės į dešinę, kol pilnai užpildomas visas lygis. Ir tik pilnai užpildžius visą lygį, pereinama į kitą lygį.

Sakykime turime masyvą:

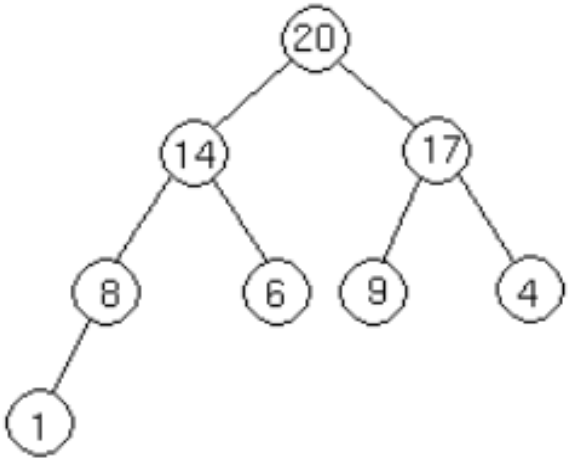
**[100; 19; 36; 17; 3; 25; 1; 2; 7 ]**

Indeksai apskaičiuojami taip:

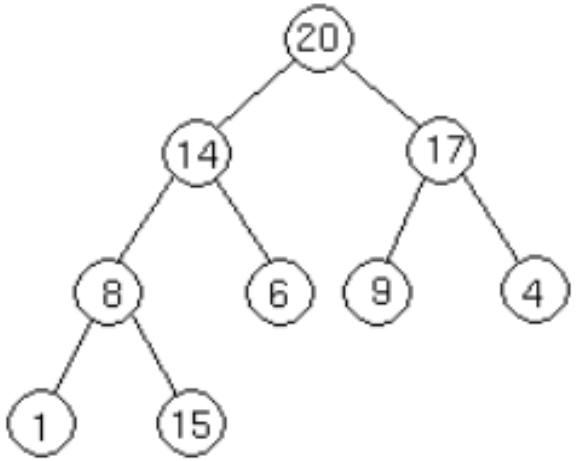
```
PARENT ( i ) = floor ( i / 2 );
LEFT ( i )     = 2 * i;
RIGHT ( i )    = 2*i + 1;
```



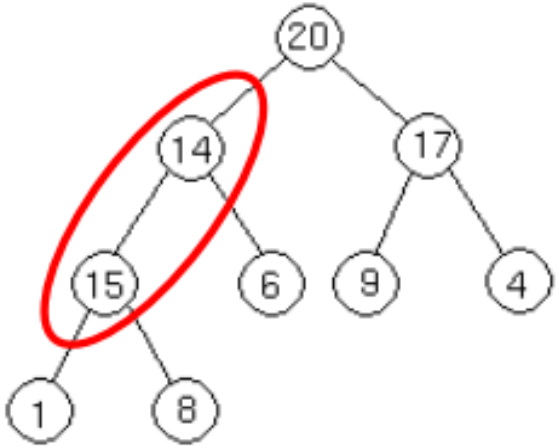
Pradinė piramidė:

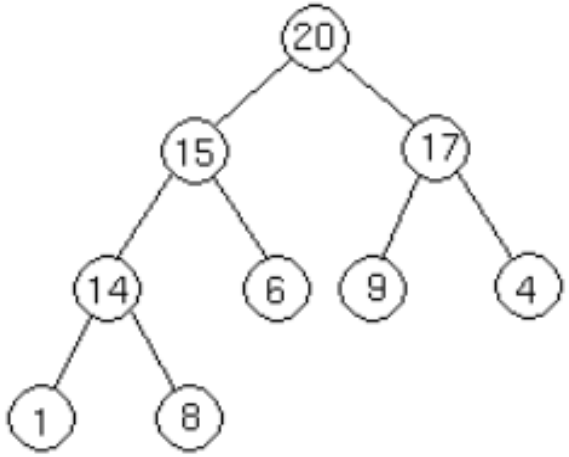



Pridedama viršūnė su reikšme 15:




Viršūnių sukeitimas:



# Piramidės rūšiavimo algoritmas

Algoritmo žingsniai:

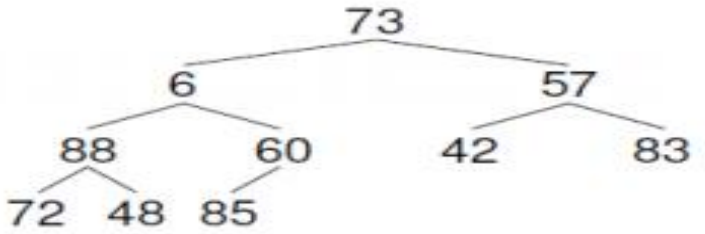
1. Piramidės rūšiavimas pradedamas sudarant dvejetainę piramidę. Po to pašalinama šaknis ir padedama į dalinai išrūšiuoto masyvo pabaigą.

2. Pašalinus šaknį, rekonstruojama piramidė **surandant kitą didžiausią elementą** ir jis taip pat pašalinamas padedant jį dalinai išrūšiuotą masyvą.

3. Tai kartojama, kol piramidėje nebelieka viršūnių, o masyvas yra pilnai išrūšiuotas. **Šiam alogritmui reikalingi du masyvai: vienas piramidei, kitas sudėti išrūšiuoto masyvo elementus.**

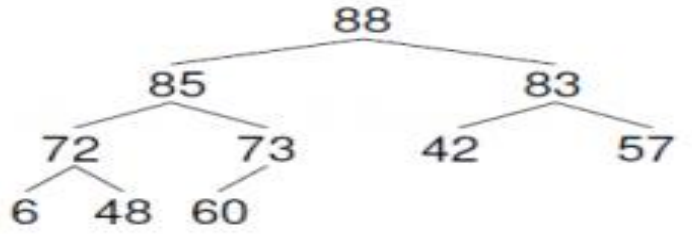
Original Numbers

| 73 | 6 | 57 | 88 | 60 | 42 | 83 | 72 | 48 | 85 |
|---|---|---|---|---|---|---|---|---|---|



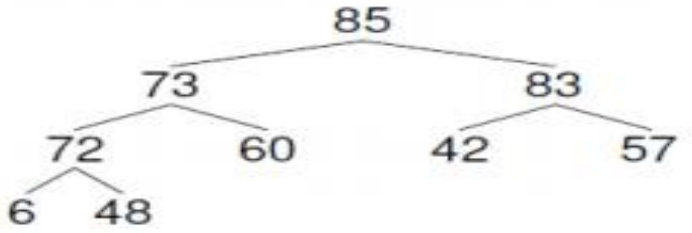

Build Heap

| 88 | 85 | 83 | 72 | 73 | 42 | 57 | 6 | 48 | 60 |
|---|---|---|---|---|---|---|---|---|---|



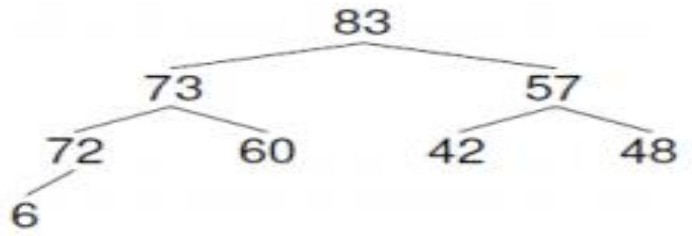

Remove 88

| 85 | 73 | 83 | 72 | 60 | 42 | 57 | 6 | 48 | 88 |
|---|---|---|---|---|---|---|---|---|---|



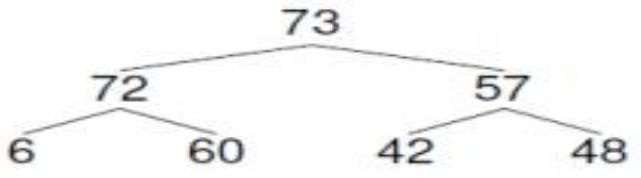

Remove 85

| 83 | 73 | 57 | 72 | 60 | 42 | 48 | 6 | 85 | 88 |
|---|---|---|---|---|---|---|---|---|---|




Remove 83

| 73 | 72 | 57 | 6 | 60 | 42 | 48 | 83 | 85 | 88 |
|---|---|---|---|---|---|---|---|---|---|



# Piramidės algoritmo realizacija

```
void HeapSort ( int array[ ], int n)
{ int i, temp;
  heap (array ,n);

  for( i = n-1; i >= 1; i-- )
  {
      temp =  array[ i ];
      array[ i ] = array[ 0 ];
      array[ 0 ] = temp;
      heap (array, i-1);
  }
}
```

```
void heap (int *a, int n)
{
 int i, temp;
 for ( i = n/2;  i >= 0;  i -- )
 {
    if ( a[(2*i) + 1]  <  a[(2*i) + 2]  &&  (2*i +1) <= n  && (2*i + 2) <= n )
    {
       temp = a[(2*i) + 1];
       a[(2*i) + 1] = a[(2*i) + 2];
       a[(2*i) + 2] = temp;
    }
   if ( a[(2*i) + 1] > a[i]  && (2*i + 1) <=n  &&  i <= n)
    {
       temp = a[(2*i) + 1];
       a[(2*i) + 1] = a[i];
       a[ i ] = temp;
    }
  }
}
```

# Piramidės algoritmo sudėtingumas

Piramidės konstravimo sudėtingumas: **T(n) = O(n);**

n kartų ištrinant šaknį reikia atlikti : **T(n) = O(log n);**

Bendras piramidės rūšiavimo algoritmo sudėtingumas blogiausiu, vidutiniu ir geriausiu atveju: **T(n) = O(n log n)**

Piramidės algoritmo privalumas tas, kad jo veiksmų skaičius nepriklauso nuo pradinės situacijos masyve.

Piramidės rūšiavimo algoritmas bendrai lėtesnis nei spartusis algoritmas, bet blogiausiu atveju jis greitesnis už spartųjį. (Quicksort T(n) = O(n2 ) blogiausiu atveju).