DUOMENŲ STRUKTŪROS IR ALGORITMAI

Greitieji rūšiavimo algoritmai (tęsinys)

Piramidės (Heap) ir Radix algoritmai

Praeitos paskaitos santrauka

- Spartieji rūšiavimo algoritmai:
  - Šelo
  - Suliejimo
  - Spartusis
- Kodėl šie algoritmai vadinami **spartieji**?
- Koks metodas naudojamas visuose sparčiuosiuose algoritmuose ir kokie jo žingsniai?

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?

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.

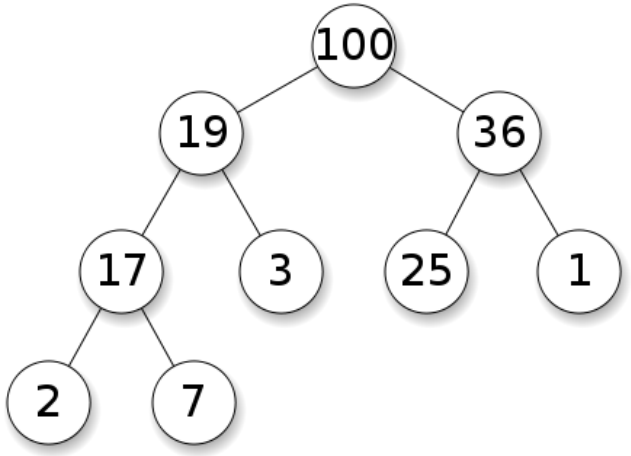
Sakykime turime masyvą:

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

Indeksai apskaičiuojami taip:

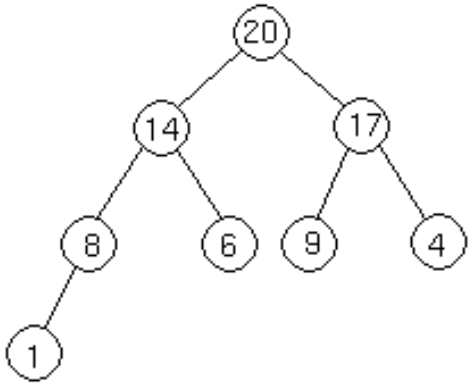
PARENT (*i*) = floor (*i* / 2);
LEFT (*i*) = 2 * *i;*
RIGHT (*i*) = 2**i* + 1;

P.S. šaknies indeksas = 1

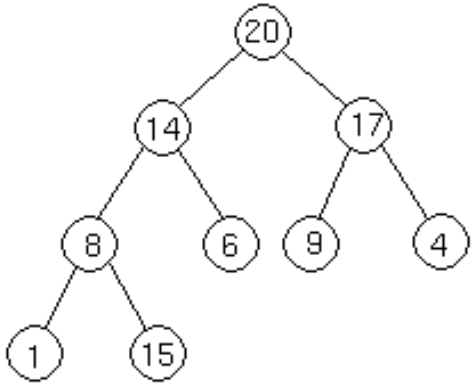


Viršūnės įterpimas

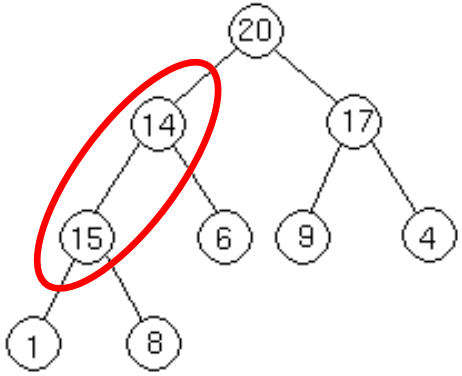
Pradinė piramidė:

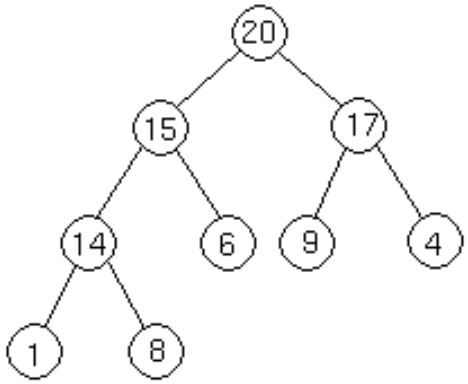


Pridedama viršūnė su reikšme 15:



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.

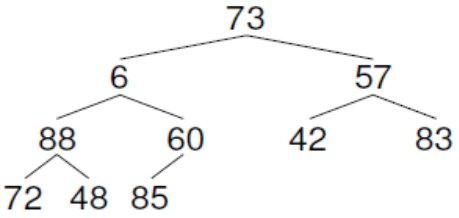
2. Pašalinus šaknį, rekonstruojama piramidė surandant kitą didžiausią elementą ir jis taip pat pašalinamas padedant jį dalinai išrūšiuotą masyvą.

3. Tai kartojama, kol piramidėje nebelieka viršūnių, o masyvas yra pilnai išrūšiuotas.

Šiam alogritmui reikalingi du masyvai: vienas piramidei, kitas sudėti išrūšiuoto masyvo elementus.

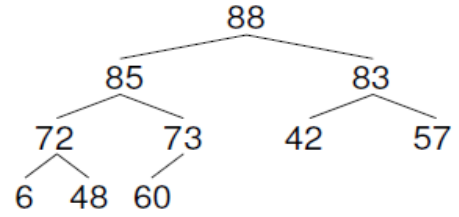
Original Numbers

73	6	57	88	60	42	83	72	48	85



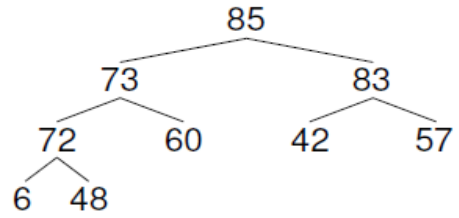
Build Heap

88	85	83	72	73	42	57	6	48	60



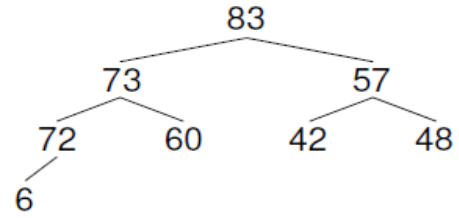
Remove 88

85	73	83	72	60	42	57	6	48	88



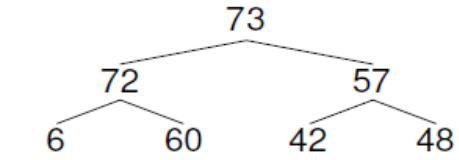
Remove 85

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);
  }
}
```

Piramidės algoritmo realizacija

```
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 $T(n) = O(n^2)$ blogiausiu atveju).

Radix rūšiavimo algoritmas

Toliau susipažinsime su specialiaisiais rūšiavimo algoritmais, kurie yra greitesni už bendruosius algoritmus, kadangi naudojasi papildoma informacija apie aibės elementus arba uždavinys tenkina papildomus apribojimus.

Radix rūšiavimo algoritmas – tai toks sveikų skaičių rūšiavimo algoritmas, kuris lygina skaitmenis esančius toje pačioje skaitmens vietoje.

Kadangi sveikais skaičiais galime išreikšti raides (pvz. ASCII kodai), todėl **Radix** algoritmas neapsiriboja vien tik sveikais skaičiais.

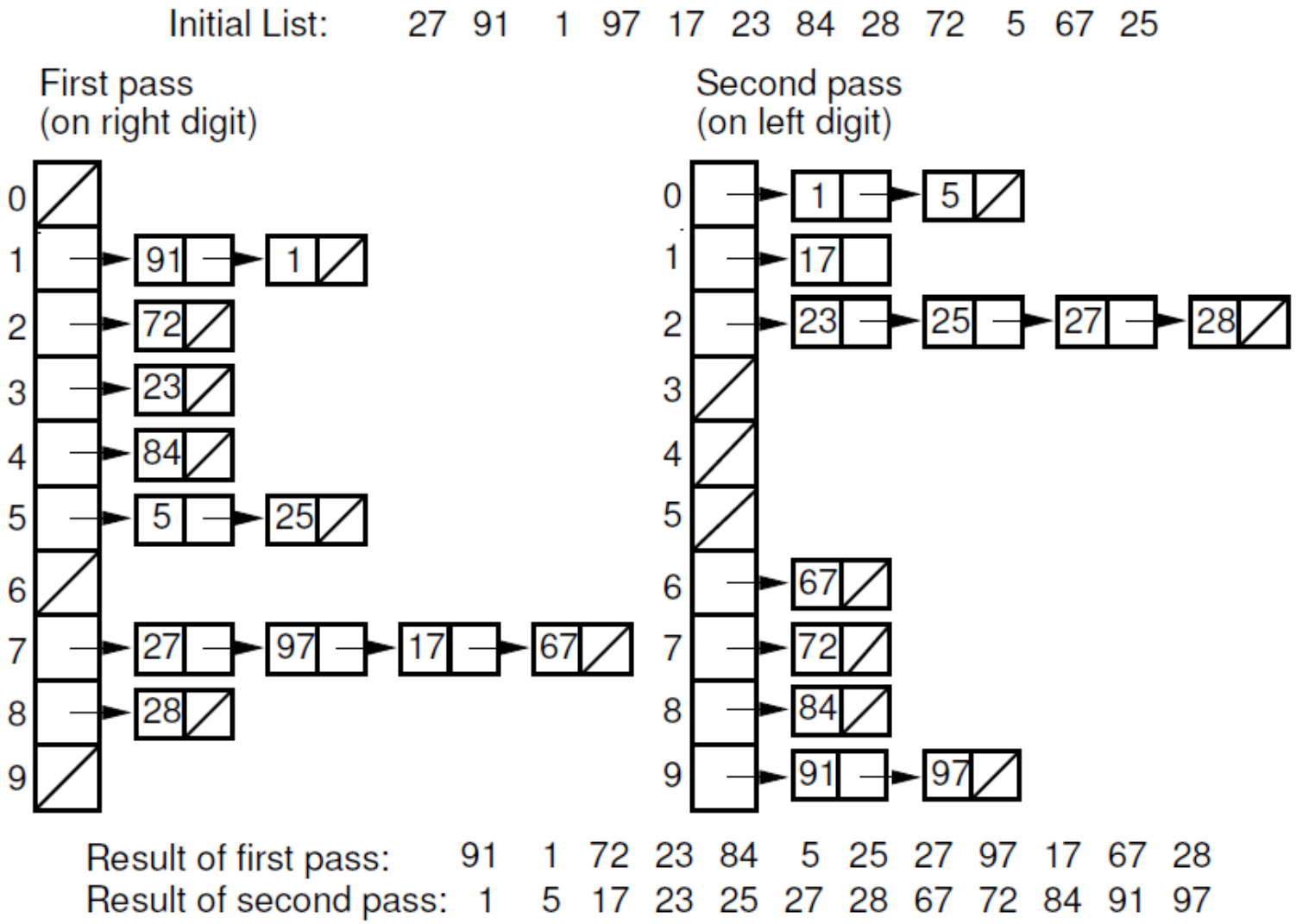
Radix rūšiavimo algoritmas

Tarkime, kad turime A masyvą, kurio elementų reikšmės yra sveiki *k*–ženkliai skaičiai. Algoritme pasinaudojame dešimtainės aritmetikos taisyklėmis.

Radix algoritmo žingsniai:

1. Visus elementus suskirstome į dešimt poaibių pagal paskutinį rakto skaitmenį (vienetų pozicija).
2. Visus poaibius jų eiliškumo tvarka sujungiame į vieną masyvą.
3. Gautąjį masyvą vėl skirstome į dešimt poaibių pagal priešpaskutinį elemento skaitmenį (dešimčių pozicija) ir surūšiuojame.
4. Procesą kartojame *k* kartų.

Radix rūšiavimo pavyzdys



Algoritmo teisingumo analizė

Imkime du skaičius **X** ir **Y**, kurie užrašomi taip:

X = 10 i + j; **Y = 10 m + n,** kur 0 <= i, j, m, n <= 9

Nelygybė **X < Y** yra teisinga jei:

i < m **arba** **i = m** **ir** ***j < n***

Radixsort algoritmas tuos žingsnius ir realizuoja, todėl algoritmo veikimo principas teisingas.

Radix algoritmo sudėtingumas

Radix algoritmas ***k*** kartų rūšiuoja masyvą, kurį sudaro ***n*** skaičių susidenančių iš ***r*** skaitmenų.
Taigi kiekvieną kartą reikia atlikti (***n*** + ***r***) veiksmų. Bendras atliekamas darbas yra:

$$T(n) = O(nk + rk).$$

Tai nebūtinai geresnis atvejis nei O(n*log(n)), nes k gali būti nepriklausomas nuo n.

Radix algoritmo sudėtingumas

Pavyzdys

Turime masyvą iš *n* skirtingų skaičių. Sakysime skaitmenys yra dešimtainiai (bendru atveju gali būti ir kitokie).

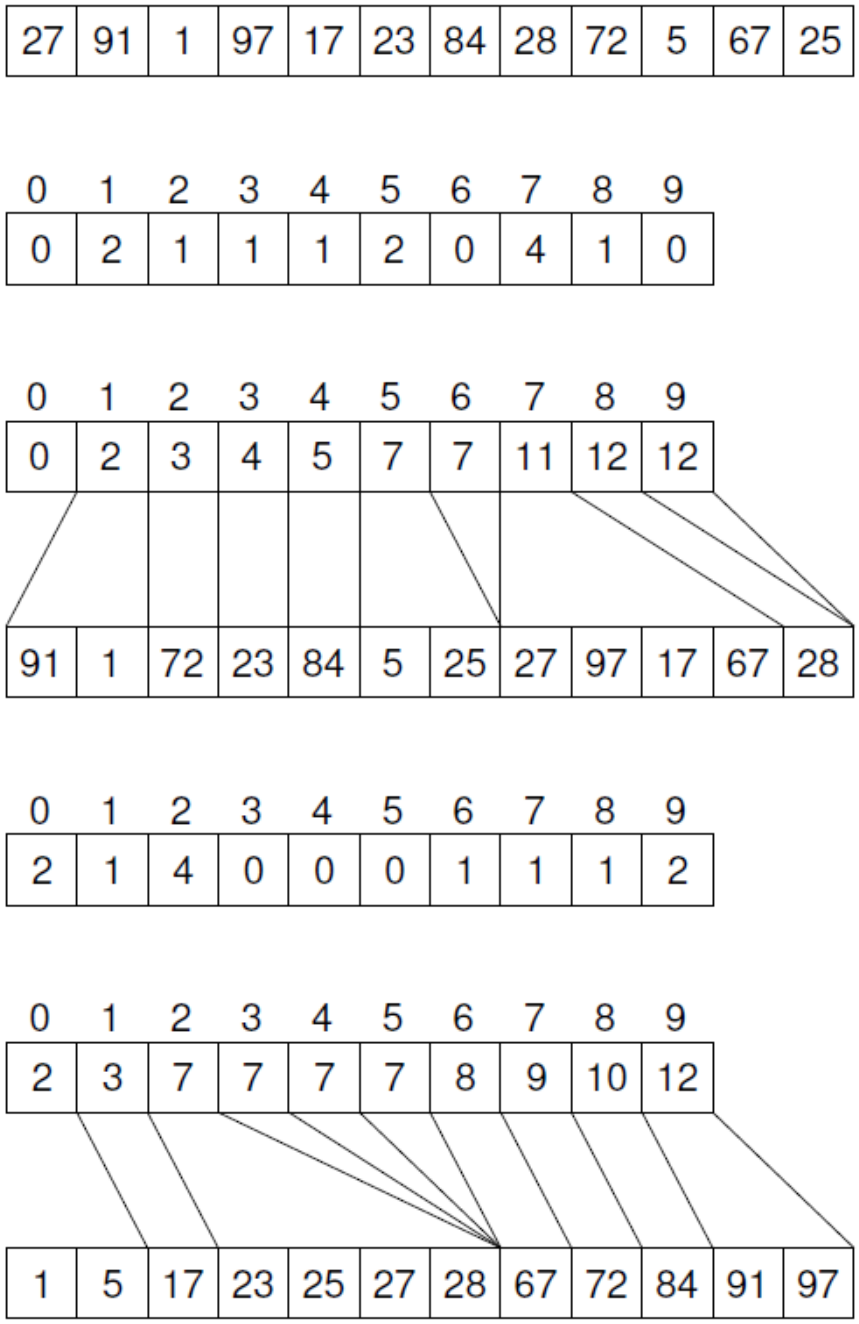
Tai reiškia, kad 10 skirtingų skaitmenų gali būti vienoje pozicijoje ir *k* gali būti ne didesnis nei $\mathbf{log_{10}(}\boldsymbol{n}\mathbf{)}$.

Kiekvienas rūšiavimo veiksmas pareikalaus vidutiniškas $n*\log_2(10)$ lyginimo ir sukeitimo veiksmų, todėl sudėtingumas bus:

$$\mathbf{T\ (n) = n*log_2(n)}$$

Radix algoritmo realizacija

```
#define MAX 10
void Radixsort (int *A, int n)
{   int i, B[MAX], m = 0, exp = 1;                  // B[MAX] – laikiniems saugojimams
            for ( i = 0; i < n; i++)                // randamas  max elementas
            {   if (A [i] > m)
                    m = A[i];
            }
            while (m / exp > 0)
            {     int bucket [10] = {0};                  // kodėl bucket [10] ?
                  for( i = 0; i < n; i++ )
                        bucket [A[i] / exp %10]++;
                  for( i = 1; i < 10; i++)
                        bucket [i] = bucket [i] + bucket[i-1];
                  for( i = n-1; i >= 0; i--)
                        B [--bucket [A[i] / exp %10] ] = A[i];
                  for( i = 0; i < n; i++)
                        A[i] = B[i];
                  exp*=10;
            }  }
```


27 91 1 97 17 23 84 28 72 5 67 25
0 1 2 3 4 5 6 7 8 9
0 2 1 1 1 2 0 4 1 0
0 1 2 3 4 5 6 7 8 9
0 2 3 4 5 7 7 11 12 12
91 1 72 23 84 5 25 27 97 17 67 28
0 1 2 3 4 5 6 7 8 9
2 1 4 0 0 0 1 1 1 2
0 1 2 3 4 5 6 7 8 9
2 3 7 7 7 7 8 9 10 12
1 5 17 23 25 27 28 67 72 84 91 97

Empyrinis lyginimas

Rūšiavimo algoritmų asimptotinės analizės trūkumai:

• Asimptotinis algoritmų vertinimas leidžia įvertinti algorimtų lėtų ir greitų algoritmų skirtumus pvz.: $O(n^2)$ and $O(n \log n)$, bet sunkiai vertina algoritmus, kurie turi tą patį sudėtingumą.

• Asimptotinė analizė nepateikia vertinimo, kuris rūšiavimo algoritmas yra geresnis dirbant mažomis aibėmis.

Empyrinio lyginimo rezultatai

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99