**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

****

**ALGORITMŲ SUDARYMAS IR ANALIZĖ**

1 laboratorinis darbas

(Varianto Nr.2)

Atliko: Aivaras Klišauskas IFK-8gr.

**KAUNAS, 2011**

**1. Rūšiavimas „InsertionSort“ algoritmu**

**1.1 Rūšiavimo „InsertionSort“ algoritmu teorinis įvertinimas**

Kiekvienos iteracijos metu elementas yra įdedamas į jau surikiuotą sąrašo dalį. Tuomet pereinamą per surikiuotąją dalį iš galo į priekį kol randama tinkama vieta tam elementui. Elementas tuomet įdedamas į tą vietą, o visi einantieji už jo yra perslenkami į kairę, būtent dėl šios priežasties, jeigu sąrašas jau yra surikiuotas, rikiavimas naudojant šį algoritmą būtų lygus O(*n*), kadangi kiekvienas elementas jau yra surikiuotas. Kitu atveju - jeigu sąrašas surikiuotas atvirkštine tvarka, rikiavimas trukų O(*n*2), kadangi jis ieškotų per visą surikiuotąją dalį, ir kiekvieną kartą įstatydamas elementą sukeistų toliau esančius.

**1.2 Rūšiavimo „InsertionSort“ algoritmu išeities tekstai**

public class InsertSorter {

protected FileArrayInt arr;

protected int n;

protected long ops;

public static void main(String[] args) {

InsertSorter bs = new InsertSorter();

try {

System.out.println("Pradedama rikiuoti...");

Timer t = new Timer(true);

bs.sort("duomenys.txt");

t.end();

System.out.println("Surikiuota.");

bs.stats();

System.out.println(t.duration() + " ms.");

} catch (Exception ex) {

System.out.println("Klaida rikiuojant.");

ex.printStackTrace();

}

}

public InsertSorter() {

ops = 0;

}

public void sort(String fileArray) throws Exception {

arr = new FileArrayInt(fileArray);

n = arr.size();

InsertSort();

arr.close();

ops += 3;

}

private void InsertSort() throws Exception {

boolean work = true;

int last = n - 1; // gali but kad nereikia -1;

for( int p = 1; p < last; p++ )

{

Comparable tmp = arr.get(p);

int j = p;

for( ; j > 0 && tmp.compareTo( arr.get(p-1) ) < 0; j-- ){

arr.set(j, arr.get(j-1)); ops += 2;}

arr.set(j, Integer.parseInt(tmp.toString()));

ops += 2;

}

}

public void stats() {

System.out.println(n + " elementu.");

System.out.println(ops + " operaciju.");

}

}

**1.3 Rūšiavimo „InsertionSort“ algoritmo eksperimentų rezultatai**

|  |  |  |
| --- | --- | --- |
| Duomenų imtis | Algoritmo atliko laikas (sekundėmis) | Operacijų kiekis |
| 100 | 709 | 9661 |
| 200 | 2008 | 29745 |
| 300 | 5885 | 88811 |
| 400 | 10493 | 157639 |
| 500 | 18343 | 246645 |

**2.Rikiavimas piramide**

**2.1 Rūšiavimo piramide algoritmo teorinis įvertinimas**

HeapSort algoritmas – kai rikiuojama duomenis sukeliant į krūvos(piramidinę) struktūrą

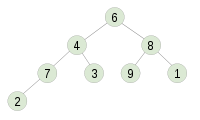
1. Sudaroma Krūvos duomenų struktūra
2. Iteratiškai iš sudarytos krūvos pašalinamos šaknys. Jas išsaugome mums reikiama tvarka: pvz., realizuojant masyvu, galime ją įrašyti į masyvo gale atsilaisvinusią vietą (pašalinus šaknį ji yra pakeičiama mažiausiu krūvos lapu).

Krūva sudaroma iš masyvo {a1, a2, ..., an}. sudaroma taip kad:

{a1', a2', ..., an'), kur ai' >= a2i' ir ai' >= a2i+1'

Antroji algoritmo dalis yra pats rikiavimas. Eilinėje iteracijoje imama krūvos viršūnė ir sukeičiama vietomis su paskutiniu masyvo elementu. Kadangi krūvos viršūnėje visada bus didžiausias elementas, tai jį sukeitę vietomis su paskutiniu masyvo elementu, žinosime, kad maksimalus masyvo elementas yra jo paskutinėje pozicijoje. Taip sukeitę elementus vietomis, turime atkurti krūvą. Tai padaryti paprasta, nes vietą pakeitęs būna tik vienas elementas (jis atsiduria viršūnėje). Šį elementą, jei jis nėra didžiausias, reikia perstumti keletu lygių žemiau, kol vėl gausime krūvą. Tai atlikę, pradedame naująją iteraciją, tik šį kartą jau nagrinėsime vienetu trumpesnį masyvą, nes didžiausias elementas, kuris anksčiau buvo viršūnėje, jau atsidūrė savo vietoje ir jo nagrinėti nereikia.

**Krūvos sudarymas:**Tarkim, turime duomenų masyvą{a1, a2, ..., an}, Pirmiausia sukeiskime jo elementus vietomis, kad gautume{a1', a2', ..., an'), kur ai' >= a2i' ir ai' >= a2i+1'.

Grafiškai tai atrodys kaip dvejetainis medis, kurio kiekvienos viršūnės sūnūs yra ne didesni už tėvus.

Tarkime, turime pradinius duomenis {6, 4, 8, 7, 3, 9, 1, 2}. Masyvo elementų skaičius n=9. Pradėsime nuo pradinio elemento i=n/2=4.

Elementų perkėlimas bus vykdomas taip:

Pav. 1 Sudarytapiramidė (krūva)

i = 4

i = 2

6 4 9 7 3 8 1 2

^ ^ ^

| |

<--->

6 7 9 4 3 8 1 2

i = 1

6 7 9 4 3 8 1 2

^ ^ ^

| |

<--->

9 7 6 4 3 8 1 2

^ ^ ^

| |

<----->

9 7 8 4 3 6 1 2

6 4 8 7 3 9 1 2

^ ^

i = 3

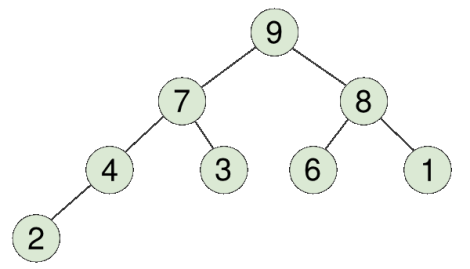
6 4 8 7 3 9 1 2

^ ^ ^

| |

<----->

6 4 9 7 3 8 1 2



Pav. 2 Surikiuotapiramidė (krūva)

Taigi gavome {9, 7, 8, 4, 3, 6, 1, 2}. Šiame masyve jau tenkinama mūsų minėta sąlyga.

Tokia duomenų struktūra vadinama [piramide (krūva)](http://lt.wikipedia.org/wiki/Kr%C5%ABva). Pirma algoritmo dalis būtent ir buvo krūvos sudarymas.

Antroji algoritmo dalis yra pats rikiavimas. Eilinėje iteracijoje imama krūvos viršūnė ir sukeičiama vietomis su paskutiniu masyvo elementu. Kadangi krūvos viršūnėje visada bus didžiausias elementas, tai jį sukeitę vietomis su paskutiniu masyvo elementu, žinosime, kad maksimalus masyvo elementas yra jo paskutinėje pozicijoje. Taip sukeitę elementus vietomis, turime atkurti krūvą. Tai padaryti paprasta, nes vietą pakeitęs būna tik vienas elementas (jis atsiduria viršūnėje). Šį elementą, jei jis nėra didžiausias, reikia perstumti keletu lygių žemiau, kol vėl gausime krūvą. Tai atlikę, pradedame naująją iteraciją, tik šį kartą jau nagrinėsime vienetu trumpesnį masyvą, nes didžiausias elementas, kuris anksčiau buvo viršūnėje, jau atsidūrė savo vietoje ir jo nagrinėti nereikia. Tokių iteracijų skaičius yra N-2.

Krūvos aukštis gali būti rastas formule \lceil \log_2 {n +1} \rceilAlgoritmo sudėtingumas: O()

**2.2 Rūšiavimo piramide algoritmo išeities tekstai**

public class HeapSorter {

protected FileArrayInt arr;

protected int n;

protected long ops;

public static void main(String[] args) {

HeapSorter hs = new HeapSorter();

try {

System.out.println("Pradedama rikiuoti...");

Timer t = new Timer(true);

hs.sort("duomenys.txt");

t.end();

System.out.println("Surikiuota.");

hs.stats();

System.out.println(t.duration() + " ms");

} catch (Exception ex) {

System.out.println("Klaida rikiuojant.");

ex.printStackTrace();

}

}

public HeapSorter() {

}

public void sort(String fileArray) throws Exception {

arr = new FileArrayInt(fileArray);

n = arr.size();

heapSort();

n = arr.size();

arr.close();

ops += 4;

}

private void heapSort() throws Exception {

buildMaxHeap();

for(int i = n; i > 0; i--) {

arr.swap(0, n - 1);

n--;

heapify(0);

ops += 6;

}

}

private void buildMaxHeap() throws Exception {

for(int i = (n - 1) / 2; i >= 0; i--) {

heapify(i);

ops += 2;

}

}

private void heapify(int i) throws Exception {

int largest;

int l = left(i);

int r = right(i);

int intI = arr.get(i);

largest = (l < n && arr.get(l) > intI) ? l : i;

if(r < n && arr.get(r) > arr.get(largest)) {

largest = r;

ops++;

}

if(largest != i) {

arr.set(i, arr.get(largest));

arr.set(largest, intI);

heapify(largest);

ops += 3;

}

ops += 8;

}

private int left(int i) {

return 2 \* i + 1;

}

private int right(int i) {

return 2 \* i + 2;

}

public void stats() {

System.out.println(n + " elementu.");

System.out.println(ops + " operaciju.");

}

}

**2.3 Rūšiavimo piramide algoritmo eksperimentų rezultatai**

|  |  |  |
| --- | --- | --- |
| Duomenų imtis | Algoritmo atliko laikas (mili sekundėmis) | Operacijų kiekis |
| 100 | 208 | 7413 |
| 200 | 654 | 16838 |
| 300 | 970 | 27560 |
| 400 | 983 | 38490 |
| 500 | 1278 | 49682 |

**3.Rikiavimas „Radix“ algoritmu**

**3.1 Rūšiavimo „Radix“ algoritmo teorinis įvertinimas**

Radix sort algoritmas yra rūšiavimo algoritmas, kuris rūšiuoja sveikus skaičius analizuodamas individualius skaitmenis. Radix sort algoritmas paprastai rūšiavimo pirmenybę:trumpi „raktai“ eina prieš ilgus raktus ir raktai, kurie yra tokio pat ilgio rūšiuojami leksikografiškai. Seka: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Radix sort algoritmas rūšiavimui naudoja leksikografinį eiliškumą, kuris yra tinkamas, žodžiams, fiksuoto ilgio sveikiem skaičiam rūšiuoti. Tokia seka kaip "b, c, d, e, f, g, h, i, j, ba" bus surūšiuota leksikografiškai taip "b, ba, c, d, e, f, g, h, i, j". Jei rūšiavimas naudojamas sveikaskaitiniam skaičiam pvz nuo 1 iki 10 tai rūšiavimas rezultate turėsim 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, trumpiausi raktai buvo palikti dešiniau tam, kad būtų greičiau aptiktas ilgiausias raktas.

**3.2 Rūšiavimo „Radix“ algoritmu išeities tekstai**

import java.util.logging.Level;

import java.util.logging.Logger;

public class RidixSorter {

protected FileArrayInt arr;

protected int n;

protected long ops;

public static void main(String[] args) {

RidixSorter hs = new RidixSorter();

try {

System.out.println("Pradedama rikiuoti...");

Timer t = new Timer(true);

hs.sort("duomenys.txt");

t.end();

System.out.println("Surikiuota.");

hs.stats();

System.out.println(t.duration() + " ms");

} catch (Exception ex) {

System.out.println("Klaida rikiuojant.");

ex.printStackTrace();

}

}

public RidixSorter() {

}

public void sort(String fileArray) throws Exception {

arr = new FileArrayInt(fileArray);

n = arr.size();

radixSort();

n = arr.size();

arr.close();

ops += 4;

}

public void radixSort(){

try {

if(arr.size() == 0)

return;

int[][] np = new int[arr.size()][2];

int[] q = new int[0x100];

int i,j,k,l,f = 0;

for(k=0;k<4;k++){

for(i=0;i<(np.length-1);i++)

{

np[i][1] = i+1;

ops+=2;

}

np[i][1] = -1;

for(i=0;i<q.length;i++)

{

q[i] = -1;

ops+=2;

}

for(f=i=0;i<arr.size();i++){

j = ((0xFF << (k << 3)) & (arr.get(i))) >> (k << 3);

if (q[j] == -1) {

l = q[j] = f;

} else {

l = q[j];

while (np[l][1] != -1) {

l = np[l][1];

ops+=1;

}

np[l][1] = f;

l = np[l][1];

}

f = np[f][1];

np[l][0] = arr.get(i);

np[l][1] = -1;

ops+=8;

}

for(l=q[i=j=0];i<0x100;i++)

for(l=q[i];l!=-1;l=np[l][1])

arr.set(j++, np[l][0]);

}

} catch (Exception ex) {

Logger.getLogger(RidixSorter.class.getName()).log(Level.SEVERE, null, ex);

}

}

public void stats() {

System.out.println(n + " elementu.");

System.out.println(ops + " operaciju.");

}

}

**3.3 Rūšiavimo „Radix” algoritmo eksperimentų rezultatai**

|  |  |  |
| --- | --- | --- |
| Duomenų imtis | Algoritmo atliko laikas (mili sekundėmis) | Operacijų kiekis |
| 100 | 85 | 20609 |
| 200 | 140 | 69261 |
| 300 | 206 | 147969 |
| 400 | 242 | 256739 |
| 500 | 325 | 395663 |

**4. Išvados:**

InsertionSort algoritmas nors ir yra greitas ir labiausia taikytinas nedideliam duomenų kiekiui, tačiau iš grafikų matyti, kad nerekomenduotinas taikyti didelių imčių netvarkingiems duomenims rikiuoti. Labiausia priklauso nuo geriausio ir blogiausio atvejo, tačiau vidutiniškai dirba netoli geriausio atvejo. Rūšiavimas piramide labiausiai priklauso nuo struktūros ir yra nestabilus. Didžiausias trūkumas, kad duomenys turi atitikti piramidės struktūrą. Radix Sort

skirtas atvejams, kai duomenų reikšmės yra skaitmeninės ir priklauso kokiam nors skaitiniam intervalui ar išsiskiria panašiomis savybėmis. Algoritmas stabilus ir labai greitas.