KAUNO TECHNOLOGIJOS UNIVERSITETAS

ALGORITMŲ SUDARYMAS IR ANALIZĖ (P170B400)

Laboratorinis darbas nr. 1

Užduoties Var. 23

Atliko: IF 4/13 gr. studentas *Šarūnas Šarakojis*

Priėmė: dėst. Vytautas Pilkauskas

Kaunas

2016

1 Rikiavimo užduotis

Palyginkite rikiavimo algoritmą, kai rikiavimo atliekamas masyve ir sąraše (t.y. galimos tik tai struktūrai būdingos operacijos). Rikiavimo algoritmas : “Bucket” sort. Naudojamos duomenų struktūros : masyvas, ir sąrašas, kurio rodyklės ir reikšmės yra atskiruose failuose.

Algoritmo teorinis įvertinimas:

Blogiausiu atveju : $O(n^2)$, vidutinis įvertinimas: $\Theta(n+k), čia\ k-kibirų\ (bucket)\ skaičius$, geriausiu atveju: $\Omega(n+k), čia\ k-kibirų\ (bucket) skaičius$. Informacija paimta iš: https://en.wikipedia.org/wiki/Bucket_sort.

Algoritmo programos išeities tekstas:

Žemiau pateikiamas algoritmas skirtas rikiuoti sąrašui, rikiavimas masyvui yra faktiškai analogiškas, tik vidinės operacijos, tokios kaip elementų sukeitimas, elemento paėmimas iš failo, indekso radimas ir pan. užtrunka skirtingus laikus.

```
@Override
public void sortList(FileDataAsListManipulator listManipulator) throws IOException
{
    Pair minAndMax = getMinAndMaxValues(listManipulator);
    int minValue = minAndMax.getFirstNumber();
    int maxValue = minAndMax.getSecondNumber();
    int bucketCount = (maxValue - minValue) / DEFAULT_BUCKET_SIZE + 1;
    int mainFileOffset = 0;
    List<FileDataAsListManipulator> buckets =
getListOfInitializedBuckets(bucketCount);

    distributeValuesToBuckets(listManipulator, buckets, minValue);
    for (int bucketIndex = 0; bucketIndex < bucketCount; bucketIndex++) {
        FileDataAsListManipulator bucket = buckets.get(bucketIndex);

        bucket.generatePointersFile("pointers" + bucketIndex + ".txt");
        SelectionSort.getInstance().sortList(bucket);
        mainFileOffset = mergeBucketOntoMainList(listManipulator, bucket,
mainFileOffset);
    }
}

private Pair getMinAndMaxValues(FileDataManipulator dataManipulator) throws
IOException {
    int firstInt = dataManipulator.getIntFromRandomAccessFile(0);
    int minValue = firstInt;
    int maxValue = firstInt;

    for (int i = 0, size = dataManipulator.getSize(); i < size; i++) {
        int current = dataManipulator.getIntFromRandomAccessFile(i);

        if (current < minValue) {
```

```
            minValue = current;
        } else {
            if (current > maxValue) {
                maxValue = current;
            }
        }
    }

    return new Pair(minValue, maxValue);
}

private List<FileDataAsListManipulator> getListOfInitializedBuckets(int
bucketCount) throws IOException {
    List<FileDataAsListManipulator> buckets = new ArrayList<>(bucketCount);

    for (int i = 0; i < bucketCount; i++) {
        ListTypeFileDataManipulator manipulator = new
ListTypeFileDataManipulator();

        manipulator.createNewRandomAccessFile("file" + i + ".txt");
        buckets.add(manipulator);
    }

    return buckets;
}

private void distributeValuesToBuckets(FileDataAsListManipulator listManipulator,
                                       List<FileDataAsListManipulator> buckets, int
minValue) throws IOException {
    for (int i = 0, size = listManipulator.getSize(); i < size; i++) {
        int value = listManipulator.getIntFromRandomAccessFile(i);

        buckets.get((value - minValue) /
DEFAULT_BUCKET_SIZE).addValueToDataFile(value);
    }
}

private int mergeBucketOntoMainList(FileDataAsListManipulator mainList,
FileDataAsListManipulator bucket,
                                    int mainFileOffset) throws IOException {
    int bucketSize = bucket.getSize();

    if (bucketSize != 0) {
        for (int i = 0; i < bucketSize; i++) {
            int bucketElement =
bucket.getIntFromRandomAccessFile(bucket.convertNodeIndexToFileIndex(i));
            int positionInMainFile =
mainList.convertNodeIndexToFileIndex(mainFileOffset);
            int mainFileElement =
mainList.getIntFromRandomAccessFile(positionInMainFile);

            if (bucketElement != mainFileElement) {
                int index = mainList.getIndexInDataFileFromValue(bucketElement);

                mainList.swapElements(positionInMainFile, index);
            }
            mainFileOffset++;
```

```
        }
    }

    return mainFileOffset;
}
```

Kadangi pats “bucket” sort algoritmas elementų nerikiuoja, jis tik juos paskirsto į atitinkamas dėžutes, todėl tų dėžučių rikiavimui naudojamas koks nors rikiavimo algoritmas, pvz: Selection Sort. Žemiau pateikiamas Selection sort algoritmas sąrašui. Masyvo rikiavimas yra faktiškai analogiškas :

```
@Override
public void sortList(FileDataAsListManipulator listManipulator) throws IOException
{
    int size = listManipulator.getSize();

    for (int i = 0; i < size - 1; i++) {
        int currentIndex = listManipulator.convertNodeIndexToFileIndex(i);
        int minIndex = currentIndex;

        for (int j = i + 1; j < size; j++) {
            int searchIndex = listManipulator.convertNodeIndexToFileIndex(j);

            if (listManipulator.getIntFromRandomAccessFile(searchIndex) <
                    listManipulator.getIntFromRandomAccessFile(minIndex)) {
                minIndex = searchIndex;
            }
        }

        if (currentIndex != minIndex) {
            listManipulator.swapElements(currentIndex, minIndex);
        }
    }
}
```

Bene sudėtingiausia operacija sąrašui yra elementų sukeitimas. Kadangi keičiamos tik rodyklės, o ne elementai, reikia modifikuoti prieš elementus esančias, po elementų esančias ir keičiamų elementų rodykles. Taip pat dar reikia atsižvelgti ar yra tarpas tarp dviejų elementų. Žemiau pateikiama elementų sukeitimo operacija sąrašui:

```
    @Override
    public void swapElements(int firstValueIndex, int secondValueIndex) throws
IOException {
        NodePointerData firstValueData = new NodePointerData(firstValueIndex);
        NodePointerData secondValueData = new NodePointerData(secondValueIndex);

        if (!areNodesAdjacent(firstValueData, secondValueData)) {
            readjustPreviousNodesIndexes(firstValueData, secondValueData);
            readjustPreviousNodesIndexes(secondValueData, firstValueData);
            readjustNextNodesIndexes(firstValueData, secondValueData);
            readjustNextNodesIndexes(secondValueData, firstValueData);
        } else {
            if (firstValueIndex < secondValueIndex) {
                if (firstValueData.getPreviousElementIndex() ==
```

```
secondValueData.getNodeIndex()) {
                    readjustPreviousNodesIndexes(secondValueData, firstValueData);
                } else {
                    readjustPreviousNodesIndexes(firstValueData, secondValueData);
                }

                if (secondValueData.getNextElementIndex() ==
firstValueData.getNodeIndex()) {
                    readjustNextNodesIndexes(firstValueData, secondValueData);
                } else {
                    readjustNextNodesIndexes(secondValueData, firstValueData);
                }

                if (firstValueData.getPreviousElementIndex() ==
secondValueData.getNodeIndex()) {
                    swapNeighborNodesIndexes(secondValueData, firstValueData);
                } else {
                    swapNeighborNodesIndexes(firstValueData, secondValueData);
                }
                return;
            } else {
                if (secondValueData.getPreviousElementIndex() ==
firstValueData.getNodeIndex()) {
                    readjustPreviousNodesIndexes(firstValueData, secondValueData);
                } else {
                    readjustPreviousNodesIndexes(secondValueData, firstValueData);
                }

                if (firstValueData.getNextElementIndex() ==
secondValueData.getNodeIndex()) {
                    readjustNextNodesIndexes(secondValueData, firstValueData);
                } else {
                    readjustNextNodesIndexes(firstValueData, secondValueData);
                }

                if (secondValueData.getPreviousElementIndex() ==
firstValueData.getNodeIndex()) {
                    swapNeighborNodesIndexes(firstValueData, secondValueData);
                } else {
                    swapNeighborNodesIndexes(secondValueData, firstValueData);
                }
                return;
            }
        }

        swapNodesIndexes(firstValueData, secondValueData);
    }
```

O tuo tarpu elementų sukeitimas sąraše yra labai paprastas ir atliekamas per pastovų laiką. Žemiau pateikiama elementų sukeitimo operacija masyvui:

```
@Override
public void swapElements(int firstValueIndex, int secondValueIndex) throws
IOException {
    int temp = getIntFromRandomAccessFile(firstValueIndex);
```

```
    addValueToDataFile(firstValueIndex,
getIntFromRandomAccessFile(secondValueIndex));
    addValueToDataFile(secondValueIndex, temp);
}
```

Algoritmo sudėtingumas:

$T\ bucketSort = O(n^2), čia\ k - kibirų\ (bucket)\ skaičius$. Elementų paskirstymas bus atliekamas masyvui pas atliekamas per panašų laiką, tačiau pats rikiavimas gerokai skirsis nes struktūros turi kitokias operacijas su elementais.

$T\ selectionSortList = O(n^2 + n)$. Rikiuoti patį sąrašą užtruks žymiai ilgiau nei masyvą, nes dauguma operacijų priklauso nuo elementų kiekio.

$T\ selectionSortArray = O(n^2)$, rikiuoti patį masyvą bus gerokai paprasčiau ir greičiau.

$T\ swapList = O(n)$, apkeitimas elementų stipriai priklauso nuo elementų kiekio jame;

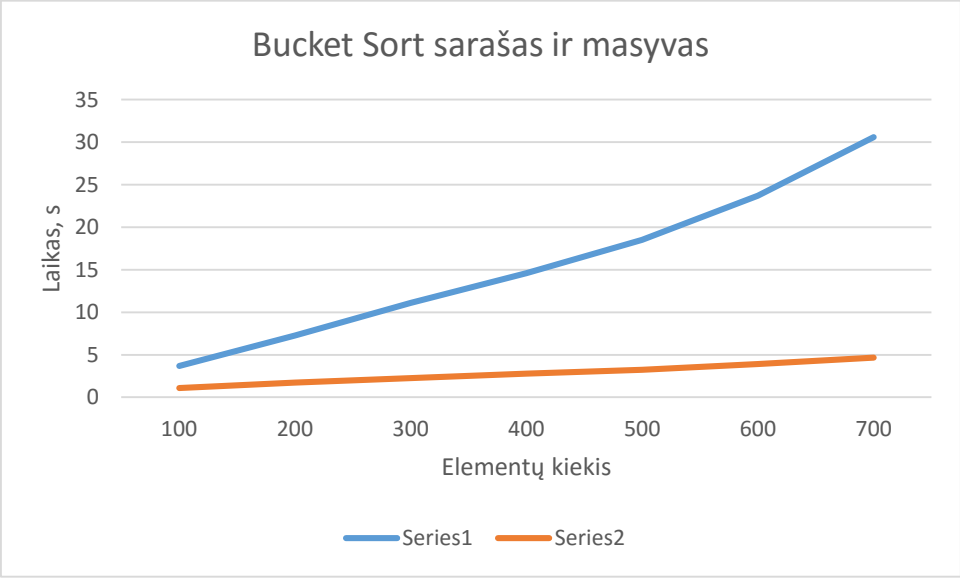
$T\ swapArray = O(1)$, tuo tarpu elementų keitimas masyve yra atliekamas per pastovų laiką.

Greitaveika:

Testai buvo atlikti su atsitiktinai sugeneruotais skaičiais. Vėliau struktūros atliko rikiavimo veiksmus su šiais duomenimis.

Elementų sk.	Sąrašas	Masyvas
100	3.67	1.099
200	7.253	1.729
300	11.104	2.241
400	14.603	2.782
500	18.511	3.231
600	23.702	3.921
700	30.576	4.679

Greitaveikos grafikas lyginant sąrašą su masyvu pateiktas žemiau:



2 Paieškos užduotis

Duotas studentų sąrašas (sugeneruokite). Suraskite sąraše vienodas pavardes. Paieškai panaudokite nurodytas duomenų struktūrą : maišos lentelė su kvadratine maišos funkcija

Duomenų struktūros teorinis įvertinimas:

Paieška: $O(1)$, vidutinis atvejis. Įterpimas: $O(1)$, vidutinis atvejis, trynimas: $O(1)$, vidutinis atvejis.

Duomenų struktūros programos išeities tekstas:

Žemiau pateikiamas algoritmas, kuris kolizijas sprendžia kvadratinės maišos funkcija:

```
@Override
public V put(K key, V value) {
    Objects.requireNonNull(key);
    Objects.requireNonNull(value);

    Node<K, V> node = new Node<>(key, value);
    index = hash(key, hashType);

    if (table[index] == null) {
        table[index] = node;
    } else {
        String line;
        boolean toAdd = true;

        try {
            conflictingNames.seek(0);

            while ((line = conflictingNames.readLine()) != null) {
                if (line.equals(key)) {
                    toAdd = false;
                    break;
                }
            }

            if (toAdd && key.equals(table[index].key)) {
                conflictingNames.seek(conflictingNames.length());
                String s = ((String) key).replaceAll("\u0000", "");
                conflictingNames.writeChars(s + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }


        for (int i = 1; i < size; i++) {
            int newIndex = (index + i * i) % size;
            if (table[newIndex] == null) {
                table[newIndex] = node;
                break;
            }
        }
    }
    size++;
```

```
    if (size > table.length * loadFactor) {
        rehash();
    }

    return value;
}
```

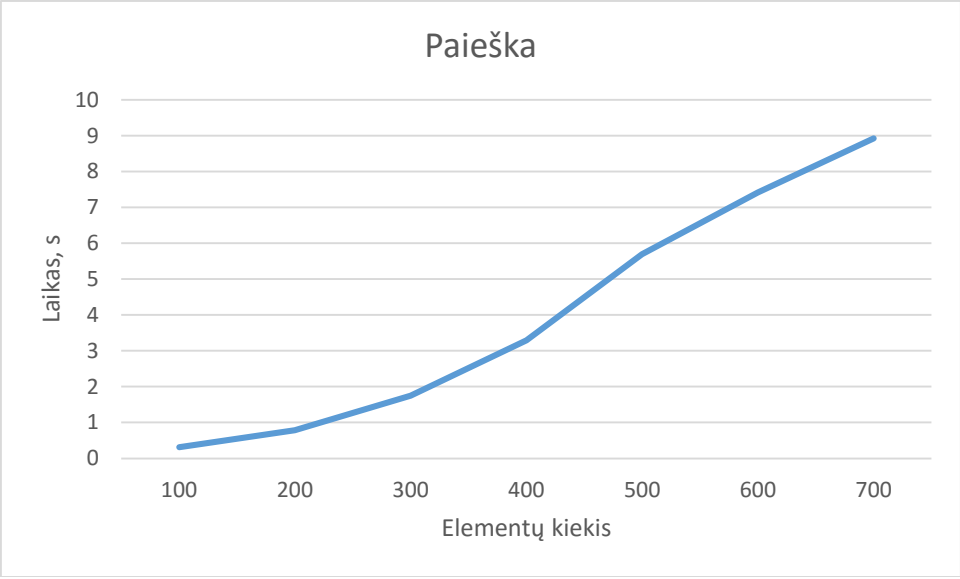
Algoritmo sudėtingumas:

$T_{add(1)}$= $O(1)$ teoriškai yra atliekamas per pastovų laiką, tačiau atsiradus kolizijai blogiausias atvejis gali būti : $O(n), čia\ n - elementų\ kiekis.$

Greitaveika:

Testai atlikti sugeneruojant atsitiktines pavardes. Pradinės pavardės saugomos išorinėje atmintyje, tai yra faile vėliau jos po vieną yra įterpiamos į lentele, jei įterpti pavardės nepavyko (atsirado kolizija), galima teigti, kad tokia pavardė jau egzistuoja lentelėje ir ji yra besikartojanti, tokiu atveju pavardė įterpiama į dublikatų sąrašą.

Elementų sk.	Lentelė (s)
100	0.31
200	0.783
300	1.745
400	3.296
500	5.691
600	7.412
700	8.921



Išvados

Parengus programą ir stebėjus kaip keičiasi greitaveika, buvo pastebėta paprasto masyvo pranašumui rikiuojant elementus. Apskritai dvikryptis sąrašas daug laiko užtrunka imdamas elementą iš sąrašo, nes reikia pereiti visus elementus nuo pačio pirmo, o tuo tarpu prie bet kurio masyvo elemento galima prieiti per pastovų laiką.

Paieškos užduotyje, atrenkant vienodas pavardes, buvo pasinaudota savybe, kad dedant tą pačią pavardę apskaičiuojamas tas pats maišos kodas ir tada galima teigti kad pavardė yra ta pati ir ją galima dėti į duplikatų sąrašą.