|  |
| --- |
| **KAUNO TECHNOLOGIJOS UNIVERSITETAS**  **INFORMATIKOS FAKULTETAS** |
| **Modulio P175B014 „Duomenų struktūros“**  Projekto aprašas (ataskaita)  **Projektas „SkipList“** |
| **Dėstytojas**  Giedrius Paulikas |
| **Studentas**  Nedas Liaudanskis |
| **KAUNAS, 2022** |

Turinys

[Realizuojamoji duomenų struktūra: 2](#_Toc120470547)

[ Duomenų tipai 2](#_Toc120470548)

[ Kitos duomenų struktūros naudojamos užduotyje 4](#_Toc120470549)

[ Esminės duomenų struktūros savybės 7](#_Toc120470550)

[ Duomenų struktūros operacijos, metodai 7](#_Toc120470551)

[Realizuotos duomenų struktūros testavimai 11](#_Toc120470552)

[Duomenų struktūros greitaveika 12](#_Toc120470553)

[ Greitaveikos testų metodika 12](#_Toc120470554)

[ Testų rezultatai 14](#_Toc120470555)

[Išvados 18](#_Toc120470556)

[ Apibendrinimas 18](#_Toc120470557)

[ Pastebėjimai 18](#_Toc120470558)

[ Palyginimas su teorija 18](#_Toc120470559)

[Literatūra 18](#_Toc120470560)

# Realizuojamoji duomenų struktūra:

## Duomenų tipai

Duomenys naudojami duomenų gali būti bet kokio tipo, tačiau testavimuose ir pačiame projekte duomenų struktūra naudos duomenų objektus Car().

Car() objektas turi tokias charakteristikas:

1. String make – kas pagamino automobilį;
2. String model – automobilio modelis;
3. Int year – pagaminimo metai;
4. Int mileage – automobilio nuvažiuotas kelias;
5. Double price – automobilio kaina;
6. String carRegNr – serijinis mašinos numeris, kiekvieną kartą bus sugeneruotas atsitiktine tvarka;
7. String idCode – serijinio numerio pradinės dvi raidės;

Duomenys projekte bus saugomi naudojant SkipList() duomenų struktūrą.

package Data;  
  
import utils.P;  
  
import java.util.InputMismatchException;  
import java.util.NoSuchElementException;  
import java.util.Random;  
import java.util.Scanner;  
  
public final class Car implements Comparable<Car>, P<Car> {  
  
 private static int *serNr* = 100;  
  
 private static final String *idCode* = "TA";  
 private final String carRegNr;  
  
 private String make = "";  
 private String model = "";  
 private int year = -1;  
 private int mileage = -1;  
 private double price = -1.0;  
  
 public Car() {  
 carRegNr = *idCode* + *serNr*++;  
 }  
  
 public Car(String dataString) {  
 Random rand = new Random();  
 carRegNr = *idCode* +( (*serNr*++) + rand.nextInt(100)); // suteikiamas originalus carRegNr  
 this.parse(dataString);  
  
 }  
  
 public Car(int codeAdder) {  
 carRegNr = *idCode* + (*serNr* + codeAdder);  
 }  
  
 public Car(String make, String model, int year, int mileage, double price, int codeadder) {  
 carRegNr = *idCode* + (*serNr* + codeadder);  
 this.make = make;  
 this.model = model;  
 this.year = year;  
 this.mileage = mileage;  
 this.price = price;  
  
 }  
  
 public String getMake() {  
 return make;  
 }  
  
 public String getModel() {  
 return model;  
 }  
  
 public int getYear() {  
 return year;  
 }  
  
 public int getMileage() {  
 return mileage;  
 }  
  
 public void setMileage(int mileage) {  
 this.mileage = mileage;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
  
 public String getCarRegNr() { //\*\* nauja.  
 return carRegNr;  
 }  
  
 @Override  
 public String toString() { // papildyta su carRegNr  
 return getCarRegNr() + "=" + make + "\_" + model + ":" + year + " " + getMileage() + " " + String.*format*("%4.1f", price);  
 }  
  
 @Override  
 public int compareTo(Car car) {  
 return getCarRegNr().compareTo(car.getCarRegNr());  
 }  
  
 @Override  
 public void parse(String dataString) {  
 try { // duomenys, atskirti tarpais  
 Scanner scanner = new Scanner(dataString);  
 make = scanner.next();  
 model = scanner.next();  
 year = scanner.nextInt();  
 setMileage(scanner.nextInt());  
 setPrice(scanner.nextDouble());  
 } catch (InputMismatchException e) {  
 e.toString();  
 } catch (NoSuchElementException e) {  
 e.getMessage();  
 }  
 }  
}

## Kitos duomenų struktūros naudojamos užduotyje

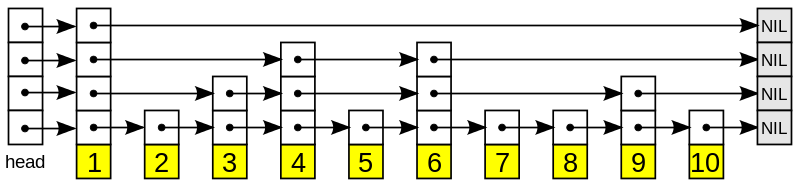
Užduotyje bus naudojama dar viena duomenų struktūra, kuri yra panašiausia SkipList() duomenų struktūrai. Ji bus naudojama greitaveikos testuose norint palyginti LinkedList() ir SkipList() duomenų struktūrų greitaveiką.

package utils;  
  
import java.nio.file.NotDirectoryException;  
  
public class LinkedList <T extends Comparable<T>> implements LinkableList<T>{  
  
 private Node<T> first;  
 private Node<T> last;  
 private Node<T> current;  
 private int size;  
  
 public LinkedList()  
 {  
  
 }  
  
 public int GetSize()  
 {  
 return size;  
 }  
  
 @Override  
 public boolean delete(T data) {  
 Node<T> pren = first;  
 if(first.data.compareTo(data) == 0)  
 {  
 first = first.next;  
 size--;  
 return true;  
 }  
 for(Node<T> n = first; n!=null; n = n.next)  
 {  
 if(n.data.compareTo(data) == 0)  
 {  
 pren.next = n.next;  
 n = null;  
 size--;  
 return true;  
 }  
 pren = n;  
 }  
 return false;  
 }  
  
 @Override  
 public String print() {  
  
 StringBuilder line = new StringBuilder();  
 line.append("\n-----------LinkedList-----------\n");  
 for(Node<T> n = first; n!= null; n = n.next)  
 {  
 line.append(n.data.toString() + "->");  
 }  
 line.append("null\n");  
  
 return line.toString();  
 }  
  
 @Override  
 public void insert(T data) {  
  
 if (data == null) {  
 throw new RuntimeException("Given element is null");  
 }  
 if(first == null)  
 {  
 first = new Node<>(data, first);  
 last = first;  
 size++;  
 }  
 else {  
  
 Node<T> node = new Node<>(data, null);  
 for(Node<T> n = first; n != null; n = n.next)  
 {  
 if(n.data.compareTo(node.data) > 0 && n == first)  
 {  
 node.next = first;  
 first = node;  
 break;  
 }  
  
 if(n.next == null)  
 {  
 n.next = node;  
 break;  
 }  
 if(n.data.compareTo(data) < 0 && n.next.data.compareTo(data) > 0)  
 {  
  
 node.next = n.next;  
 n.next = node;  
 size++;  
 break;  
 }  
 }  
 }  
 }  
  
 @Override  
 public T search(T data) {  
 for(Node<T> n = first; n!= null; n = n.next)  
 {  
 if(data.compareTo(n.data) == 0)  
 {  
 return n.data;  
 }  
 }  
 return null;  
 }  
  
 private static class Node<T> {  
 private final T data;  
 private Node<T> next;  
  
 Node(T data, Node<T> next)  
 {  
 this.data = data;  
 this.next = next;  
 }  
 }  
}

## Esminės duomenų struktūros savybės

SkipList() duomenų struktūros pagrindas yra LinkedList() duomenų struktūra. Kiekvienas SkipList() duomenų struktūros elementas yra mazgas, turintis tam tikrą informaciją ir rodyklę į kitą, toliau einantį mazgą. SkipList() struktūra sudaro mazgų, su tais pačiais duomenimis, masyvai, su atsitiktinai sugeneruotais lygiais, kuo didesnis lygis tuo daugiau mazgu bus prijungta prie šio masyvo. Galime įsivaizduoti šiuos lygius kaip, greitkelius, kuo aukštesnis lygmuo, tuo retesni duomenys. Jų pagalba, pradedant nuo aukščiausio lygio ir einant žemyn per visus lygius iki tol kol rasime mums reikiamus duomenis galime pasiekti juos daug greičiau, nes nereikia pereiti per visų duomenų struktūroje esančius elementus, o tik keletą elementų, kurie yra pasiskirstę skirtinguose lygiuose. Kiek lygių gauna elementas dažniausiai yra apskaičiuojama metant monetą: 50% jog bus vienu lygiu daugiau ir 50 % jog pasiliksime dabartiniame lygyje. Todėl ši duomenų struktūra yra sugeneruojama atsitiktine tvarka. Teoriškai ši duomenų struktūra, turėtų būti greitesnė už LinkedList() visais savo metodais:

|  |  |  |  |
| --- | --- | --- | --- |
| Metodų asimptotiniai sudėtingumai | | | |
| Metodai | LinkedList() | SkipList() | |
| Visais atvejais | Blogiausiu atveju | Dažniausiai pasitaikančiu atveju |
| Insert() | O(n) | O(n) | O(log(n)) |
| Search() | O(n) | O(n) | O(log(n)) |
| Delete() | O(n) | O(n) | O(log(n)) |



## Duomenų struktūros operacijos, metodai

Duomenų struktūra atlieka 3 pagrindines operacijas: Insert(), Search(), Delete() ir vieną papildomą Print() operacija, kurios pagalba gana lengvai yra atspausdinami esami duomenys duomenų struktūroje. SkipList() taip pat turi, pagalbinį metodą randomLevel(), kuris sugeneruoja atsitiktinį lygio aukštį.

### Insert()

Metodas sukuria naują masyvą, kuris laikys paskutinius elementus, kuriuos aplankė tam tikrame aukšte. Jis padės prisiminti, su kokiais elementais bus sujungtas naujai prijungtas elementas. Norint padaryti paiešką elemento prieš kurį dėsime, reikia pradėti nuo aukščiausio lygio, kuriame mažiausiai elementų ir tikrinti ar kitas elementas tame aukšte nėra didesnis už tą, kurį norime į dėti į duomenų struktūrą, jeigu ne, tai į masyvą įdedame aplankytą elementą ir einame prie sekančio elemento. Tai vyksta tol kol randame sekantį elementą, esamame aukšte, kuris yra didesnis už elementą, kurį reikia įterpti į duomenų struktūrą. Kai elementas yra didesnis, einame į žemesnius lygmenis, tol kol pasiekiame paskutinį lygmenį – 0, tada paieškos ciklas nustoja eiti.

Suradus vietą, kurioje turi būti įterptas naujas elementas, reikia perstatyti rodykles, tam naudojame update masyvą, tačiau jis reikalingas tada jeigu naujo elemento levelis bus aukštesnis už 0. Su jo pagalba yra gan lengvai perstatomos rodyklės ir įterpimas baigiamas.

@Override  
public void insert(T data) {  
 SkipNode<T> current = header;  
 SkipNode<T>[] update = (SkipNode<T>[]) new SkipNode[MaxLvl + 1];  
  
 for(int i = currentLevel; i >= 0; i--)  
 {  
 while(current.next[i] != null && current.next[i].data.compareTo(data) < 0)  
 {  
 current = current.next[i];  
 }  
 update[i] = current;  
 }  
  
 current = current.next[0];  
 if((current == null) || current.data != data){  
 int randomLevel = randomLevel();  
  
 if(randomLevel > currentLevel)  
 {  
 for(int i = currentLevel + 1; i < randomLevel + 1; i++)  
 {  
 update[i] = header;  
  
 }  
 currentLevel = randomLevel;  
 }  
  
 SkipNode<T> n = createNode(data, randomLevel);  
  
 for(int i = 0; i <= randomLevel; i++)  
 {  
 n.next[i] = update[i].next[i];  
 update[i].next[i] = n;  
 }  
  
 }  
}

### Search()

Paieškos algoritmas veikia panašiai kaip ir Insert() algoritmas. Yra pradedama nuo aukščiausio levelio, pirmiausia ieškoma jame, tol kol sekantis elementas nebus didesnis už tą, kurio ieškome. Jeigu jis yra didesnis, einame į žemesnį lygmenį. Tai kartojame tol, kol pasiekiame paskutinį lygmenį 0, jeigu elementas yra rasta, gražina rastą elementą. Jeigu ne, tada gražina null;

@Override  
public T search(T data) {  
 SkipNode<T> current = header;  
 for(int i = currentLevel; i >= 0; i--)  
 {  
 while(current.next[i] != null && current.next[i].data.compareTo(data) <= 0)  
 {  
 current = current.next[i];  
 }  
 }  
 if(current.data.compareTo(data) == 0)  
 {  
 return current.data;  
 }  
 return null;  
}

### Delete()

Šalinimo algoritmas veikia beveik identiškai kaip įterpimo algoritmas. Metodas sukuria naują masyvą, kuris laikys paskutinius elementus, kuriuos aplankė tam tikrame aukšte. Jis padės prisiminti, su kaip reikės sudėlioti rodykles, kaip įvyks pašalinimas. Norint padaryti paiešką elemento kurį šalinsime, reikia pradėti nuo aukščiausio lygio, kuriame mažiausiai elementų ir tikrinti ar kitas elementas tame aukšte nėra didesnis už tą, kurį norime pašalinti iš duomenų struktūros, jeigu ne, tai į masyvą įdedame aplankytą elementą ir einame prie sekančio elemento. Tai vyksta tol kol randame sekantį elementą, esamame aukšte, kuris yra didesnis už elementą, kurį reikia pašalinti. Kai elementas yra didesnis, einame į žemesnius lygmenis, tol kol pasiekiame paskutinį lygmenį – 0, tada paieškos ciklas nustoja eiti.

Suradus sekantį elementą kurį reikia pašalinti, reikia perstatyti rodykles, tam naudojame update masyvą, tačiau jis reikalingas tada jeigu naujo elemento levelis bus aukštesnis už 0. Su jo pagalba yra gan lengvai perstatomos rodyklės. Perstumdžius rodykles šalinimas yra baigiamas, nes nebėra prieigos prie šalinto elemento, jo reikšmė nustatoma į 0.

@Override  
public boolean delete(T data) {  
 SkipNode<T> current = header;  
 SkipNode<T>[] update = (SkipNode<T>[]) new SkipNode[MaxLvl + 1];  
  
 for(int i = currentLevel; i >= 0; i--)  
 {  
 while(current.next[i] != null && current.next[i].data.compareTo(data) < 0)  
 {  
 current = current.next[i];  
 }  
 update[i] = current;  
 }  
  
 current = current.next[0];  
  
  
  
 if(current.data == data)  
 {  
 for(int i = 0; current.level >= i; i++)  
 {  
  
  
 if(update[i].next[i] != current)  
 {  
 break;  
 }  
 update[i].next[i] = current.next[i];  
  
 }  
 while(currentLevel > 0 && header.next[currentLevel] == null)  
 {  
 currentLevel = currentLevel - 1;  
 }  
 return true;  
 }  
 return false;  
}

### Print()

Metodas pereina per visus lygius pradedant nuo 0 iki MaxLvl ir išrašo visus esamus elementus į konsolę, su jiems atitinkamai lygiais. Vienu žodžiu, šis metodas praeina pro visą duomenų struktūrą ir visų lygių duomenis.

@Override  
public void print() {  
 System.*out*.println("\n-----------SkipList-----------");  
 for(int i = 0; i <= currentLevel; i++)  
 {  
 SkipNode<T> node = header.next[i];  
 System.*out*.println("Level " + i + ": ");  
 while(node != null)  
 {  
 System.*out*.println(node.data + " ");  
 node = node.next[i];  
 }  
 System.*out*.println("");  
 }  
}

### randomLevel()

Metodas apskaičiuoja atsitiktinai reikšmę tarp 1 ir 0. Ta reikšmė yra tarsi monetos metimas, jeigu reikšmė yra mažesnė arba lygi 0,5 tada tas elementas, turės vienu lygiu aukštesnį lygį. Jeigu ne tada lygio šiam elementui nebereikia kelti. Šis metodas suteikia šiai duomenų struktūras atsitiktinumo.

private int randomLevel()  
{  
 double r = Math.*random*();  
 int lvl = 0;  
 while(r <= p && lvl < MaxLvl)  
 {  
 lvl ++;  
 r = Math.*random*();  
 }  
 return lvl;  
}

# Realizuotos duomenų struktūros testavimai

SkipList() duomenų struktūra testavimai buvo atlikti su keletą Car() elementų, kurių serijos numeriai buvo sugeneruoti atsitiktine tvarka. Pirmiausia visi sugeneruoti Car() elementai buvo surašyti į SkipList() naudojant Insert() metodą, tada keli iš jų pašalinti naudojant Delete() metodą ir keli surasti naudojant Search() metodą. Po kiekvieno metodo SkipList() duomenų struktūra buvo išspausdinta, todėl sekti veikimo procesą buvo gan lengvą. Po visų operacijų, buvo labai aišku kas buvo panaikinta, įdėta ar surasta. Visos operacijos po testavimų veikė be klaidų.

Po šių paprastų veikimo testų buvo atliktas ir greitaveikos testavimas, lyginant SkipList() metodų laikus su LinkedList() metodų laikais, kad tikrai būtų užtikrintas duomenų struktūros veikimas. Šis greitaveikos testavimas sugeneruodavo norimą skaičius Car() elementų, kuriuos sudėdavo į masyvą. Kai prasidėdavo testas visi šie elementai buvo naudojami atlikti 3 pagrindines SkipList() funkcijas, Insert(), Search(), Delete(). Kiekvienai funkcijai buvo užregistruotas atlikimo greitis. Su tais pačiais duomenimis visa tai buvo pakartota su LinkedList().

# Duomenų struktūros greitaveika

## Greitaveikos testų metodika

Greitaveikos testai buvo atlikti 5 iteracijomis, kuriose skyrėsi tik elementų kiekis(2000, 4000, 8000, 16000, 32000).

package Data;  
import utils.\*;  
import java.util.Arrays;  
import java.util.Collections;  
import java.util.Locale;  
import java.util.Random;  
import java.io.\*;  
  
public class SimpleBenchmark {  
  
 Car[] cars;  
 LinkedList<Car> carLinkedList;  
 SkipList<Car> carSkipList;  
 Random rg = new Random();  
 int[] counts = {2\_000, 4\_000, 8\_000, 16\_000, 32\_000};  
 void generateCars(int count) {  
 String[][] makesAndModels = { // galimų automobilių markių ir jų modelių masyvas  
 {"Mazda", "121", "323", "626", "MX6"},  
 {"Ford", "Fiesta", "Escort", "Focus", "Sierra", "Mondeo"},  
 {"Saab", "92", "96"},  
 {"Honda", "Accord", "Civic", "Jazz"},  
 {"Renault", "Laguna", "Megane", "Twingo", "Scenic"},  
 {"Peugeot", "206", "207", "307"}  
 };  
 cars = new Car[count];  
 rg.setSeed(2017);  
 for (int i = 0; i < count; i++) {  
 int makeIndex = rg.nextInt(makesAndModels.length); // markės indeksas 0..  
 int modelIndex = rg.nextInt(makesAndModels[makeIndex].length - 1) + 1;// modelio indeksas 1..  
 cars[i] = new Car((makesAndModels[makeIndex][0] + makesAndModels[makeIndex][modelIndex] + Integer.*toString*(1994 + rg.nextInt(20)) + Integer.*toString*(6000 + rg.nextInt(222\_000)) + Double.*toString*(1000 + rg.nextDouble() \* 350\_000)));  
 }  
 Collections.*shuffle*(Arrays.*asList*(cars));  
 carLinkedList = new LinkedList<>();  
 carSkipList = new SkipList<>(0.5);  
 }  
 void generateAndExecute(int elementCount) {  
 generateCars(elementCount);  
 double t0 = System.*nanoTime*();  
 for(Car car : cars)  
 {  
 carLinkedList.insert(car);  
 }  
  
 double t1 = System.*nanoTime*();  
 for(Car car : cars)  
 {  
 carLinkedList.search(car);  
 }  
 double t2 = System.*nanoTime*();  
 for(Car car : cars)  
 {  
 carLinkedList.delete(car);  
 }  
 double t3 = System.*nanoTime*();  
 //----------------------------  
 for(Car car : cars)  
 {  
 carSkipList.insert(car);  
 }  
  
  
 double t4 = System.*nanoTime*();  
 for(Car car : cars)  
 {  
 carSkipList.search(car);  
 }  
 double t5 = System.*nanoTime*();  
 for(Car car : cars)  
 {  
 carSkipList.delete(car);  
 }  
 double t6 = System.*nanoTime*();  
  
 System.*out*.println(new String("-".repeat(100)));  
 System.*out*.println("\nElement count: " + elementCount + "\nLinked list tests:\n ");  
 System.*out*.println("Insert: " + Double.*toString*((t1-t0)/1e7)+ "\nSearch: " + Double.*toString*((t2-t1)/1e7) + "\nDelete: " + Double.*toString*((t3-t2)/1e7));  
 System.*out*.println("\nSkipListtests:\n ");  
 System.*out*.println("Insert: " + Double.*toString*((t4-t3)/1e7)+ "\nSearch: " + Double.*toString*((t5-t4)/1e7) + "\nDelete: " + Double.*toString*((t6-t5)/1e7));  
  
 }  
  
 void execute() {  
 long memTotal = Runtime.*getRuntime*().totalMemory();  
 // Pasižiūrime kaip generuoja automobilius (20) vienetų)  
 generateCars(20);  
 for (Car c : cars) {  
 System.*out*.println(c);  
 }  
  
 for (int n : counts) {  
 generateAndExecute(n);  
 }  
  
 }  
 public static void main(String[] args) {  
 // suvienodiname skaičių formatus pagal LT lokalę (10-ainis kablelis)  
 Locale.*setDefault*(new Locale("LT"));  
 new SimpleBenchmark().execute();  
 }  
}

## Testų rezultatai

Greitaveika buvo atlikta du kartus, su dideliais duomenų kiekiais ir dar kartą su nedideliais duomenų kiekiais.

Greitaveikos metodų laiko palyginimo diagramos (su dideliais duomenų kiekiais)(SkipList MaxLevels: 15)(Laikas, 1\*10-2s):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Elementų kiekiai** | **LinkedList()** | | | **SkipList()** | | |
| **Insert()** | **Search()** | **Delete()** | **Insert()** | **Search()** | **Delete()** |
| 2000 | 5,14 | 1,47 | 1,36 | 0,42 | 0,38 | 0,43 |
| 4000 | 7,85 | 3,54 | 2,07 | 0,20 | 0,17 | 0,19 |
| 8000 | 39,06 | 18,76 | 11,28 | 0,55 | 0,36 | 0,39 |
| 12000 | 110,05 | 47,23 | 28,72 | 0,88 | 0,62 | 0,80 |
| 16000 | 210,96 | 87,83 | 49,73 | 0,94 | 0,83 | 1,40 |
| 20000 | 333,54 | 159,84 | 93,36 | 1,30 | 1,25 | 1,20 |
| 24000 | 385,20 | 213,35 | 136,16 | 1,97 | 1,75 | 1,46 |
| 28000 | 480,18 | 273,47 | 206,39 | 2,07 | 2,05 | 1,76 |
| 32000 | 725,59 | 444,92 | 278,29 | 2,94 | 3,00 | 3,09 |

Greitaveikos metodų laiko palyginimo diagramos (su mažais duomenų kiekiais)(SkipList MaxLevels: 15):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Elementų kiekiai** | **LinkedList()** | | | **SkipList()** | | |
| **Insert()** | **Search()** | **Delete()** | **Insert()** | **Search()** | **Delete()** |
| 5 | 0,06 | 0,0022 | 0,0014 | 0,01 | 0,0025 | 0,0023 |
| 100 | 0,5 | 0,20 | 0,13 | 0,06 | 0,08 | 0,07 |
| 200 | 0,23 | 0,12 | 0,08 | 0,04 | 0,03 | 0,03 |
| 300 | 0,04 | 0,02 | 0,01 | 0,03 | 0,04 | 0,04 |
| 400 | 0,13 | 0,06 | 0,04 | 0,04 | 0,02 | 0,02 |
| 500 | 0,12 | 0,06 | 0,04 | 0,03 | 0,02 | 0,02 |
| 600 | 0,28 | 0,12 | 0,08 | 0,05 | 0,03 | 0,03 |
| 700 | 0,27 | 0,10 | 0,06 | 0,03 | 0,02 | 0,02 |
| 800 | 0,25 | 0,12 | 0,07 | 0,03 | 0,02 | 0,03 |

# Išvados

## Apibendrinimas

Projektas pavyko tikrai gerai, programa veikia puikiai, greitaveika atitinka visus teorijos teigtus asimptotinius sudėtingumus. Iš pateiktų greitaveikos testų gan lengva pastebėti, jog SkipList() duomenų struktūra yra daug greitesnė už LinkedList() duomenų struktūrą, ypač kai didėja duomenų kiekiai. Padaręs projektą gerai išmokau šią gan greitą duomenų struktūrą, kuria galėsiu panaudoti savo ateinančiuose projektuose ar užduotyse.

## Pastebėjimai

Pastebėjau, jog greitaveika SkipList() duomenų struktūros priklauso nuo to, kaip kokiais lygiais susideda elementai. Jeigu visi elementai yra viename lygmenyje, sudėtingumas bus O(n), tačiau jeigu pavyksta juo gerai išskirstyti pro kuo daugiau lygmenų, tuo greičiau ir geriau ši duomenų struktūra veikia.

## Palyginimas su teorija

Lyginant rezultatus su teorija, viskas atitinka. LinkedList() duomenų struktūros metodų atlikimo laikas yra daug lėtesnis palyginus su SkipList() duomenų struktūra. Sudėtingumai teorijoje tikrai nemeluoja, LinkedList() visų metodų sudėtingumai O(n), tuo tarpu SkipList() O(log(n)).

# Literatūra

* <https://www.youtube.com/watch?v=2g9OSRKJuzM&t=2817s>
* <https://www.geeksforgeeks.org/skip-list/>
* <https://www.javatpoint.com/skip-list-in-data-structure>
* <https://drum.lib.umd.edu/bitstream/handle/1903/544/CS-TR-2286.1.pdf?sequence=2%20>