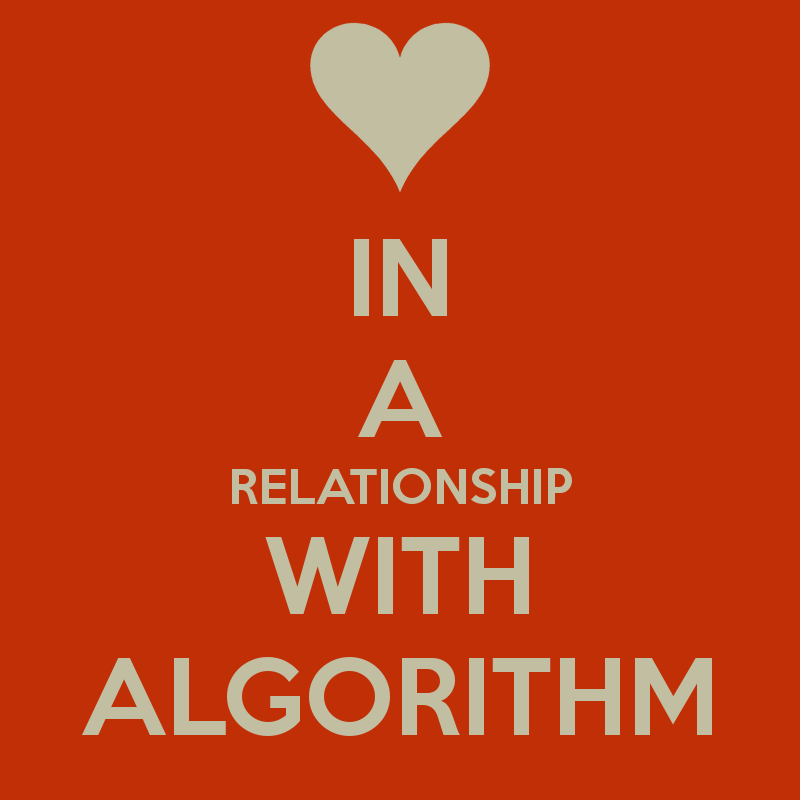
**Datastructure**

**Practicum2A-B**



Klas: IS201

Naam: Jeffrey, Alwin

Inhoudsopgave

**Inleiding3**

**Class Diagram4**

**Generate Class**5

**De SortCijfer Class**7

**BucketSort Class**9

**Sort KlasStudent** 12

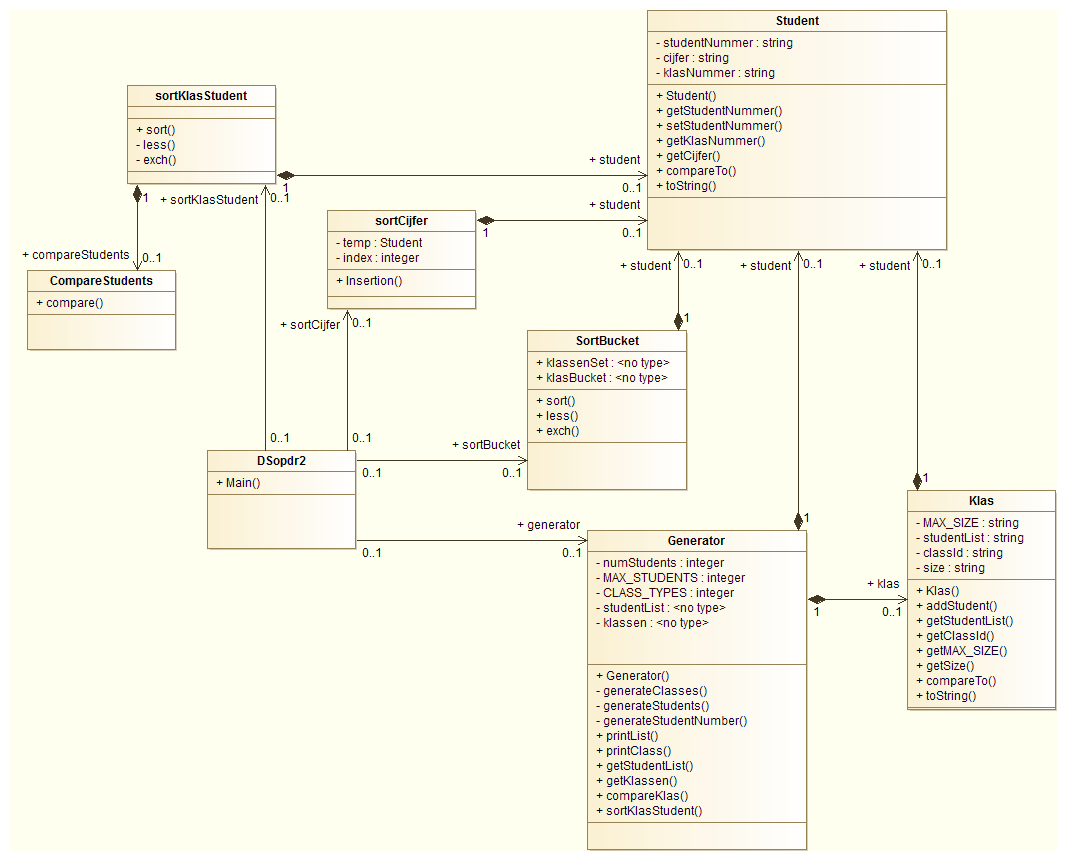
**Conclusie**13

**Inleiding**

We hebben een applicatie geschreven die een lijst met een aantal van 200, 400, 800, 1600 en 3200 studenten moet kunnen generen. Deze leerlingen krijgen allemaal een unieke studentennummer, met een cijfer van 1 tot 10 en een klas naam waarin de student uiteindelijk wordt ingedeeld. Er zijn 4 klassen richtingen IG, IS, IN of IT. Het applicatie moet een leerlingen lijst kunnen generen met behulp van een insertion sort gesorteerd op cijfers van 10 naar 1 van hoog naar laag. En het applicatie moet een lijst studenten genereren met een bucket sort gesorteerd op klassen in alfabetische volgorde en vervolgens sorteren op studentennummer per klas.

In het verslag kom 10 test resultaten van de insertion en bucket sort van toepassing, met daarbij de uiteindelijke conclusie.

**Class Diagram**



# De Generator class

Met de generate class worden alle studenten gegenereerd en in klassen ingedeeld.

Eerst word in de constructor de generateStudents methode aangeroepen. Hier wordt het meegegeven int getal gebruikt om studenten te genereren door de index van de for loop op te tellen bij 5006001.

private void generateStudents() {

for (int i = 0; i < numStudents; i++) {

studentList.add(new Student(generateStudentNumber(i), generateGrade()));

}

}

private int generateStudentNumber(int i) {

return 5006001 + i;

}

private String generateGrade() {

float result = (float) (1 + (Math.random() \* 9));

if (result > 9.95) {

return String.format("%.0f", result);

}

return String.format("%.1f", result);

}

De studentenLijst wordt geshuffled zodat bij het indelen van klassen de studenten nummers meer uiteen lopen. Daarna worden de klassen gegenereerd. De studenten worden gelijk verdeeld over de klassen van de verschillende richtingen.

//create a klas arraylist and add new classes with classname

klassen = new ArrayList<>();

for (int i = 0; i < CLASS\_TYPES; i++) {

for (int j = 0; j < numClasses; j++) {

String className = CLASS\_TYPE[i] + CLASS\_NUMBER[i];

Klas klas = new Klas(className);

klassen.add(klas);

CLASS\_NUMBER[i]++;

}

}

//add students to the classes(fills each class until the studentsPerClass is reached then moves on to the next class.

int studentCounter = 0;

for (int i = 0; i < numTotalClasses; i++) {

if (klassen.get(i).getSize() == 0) {

while (klassen.get(i).getSize() < studentsPerClass) {

klassen.get(i).addStudent(studentList.get(studentCounter));

studentCounter++;

}

}

}

Bij het afronden blijven soms studenten over die niet ingedeeld worden. Deze worden later alsnog ingedeeld bij de eerste klassen, waardoor het niet geheel eerlijk verdeeld is.

Bijvoorbeeld:

Met 400 heb je 16 klassen 4 per richting.  
100 studenten per richting en 25 per klas.

Met 397 heb je 16 klassen 4 per richting.  
96 studenten per richting en 24 per klas.  
het totaal is dan 384 als het eerlijk verdeeld is. Maar dan heb je 13 studenten over.

//add rest students to classes

for (int i = 0; i < restSumStudents; i++) {

klassen.get(i).addStudent(studentList.get(studentCounter+i));

}

Omdat we 200, 400, 800, 1600, 3200 studenten hebben komt dit niet aan de orde.

# De SortCijfer class:

temp(student) vergeleken met de vorige student in compareTo method van de Class Student.

Als temp hoger is dan verplaats het vorig student een index hoger, 'index' wordt steeds lager en de getallen worden steeds hoger totdat temp een getal tegen komt die hoger is dan hem zelf.

public class sortCijfer {

private static Student temp = null;

private static int index = 0;

public static void Insertion(ArrayList<Student> sort){

for (int i = 0; i < sort.size() ; i++) {

temp = sort.get(i);

index = i;

while (index > 0 && temp.compareTo(sort.get(index - 1)) > 0)

{

sort.set(index, sort.get(index - 1));

index --;

}

sort.set(index, temp);

}

}

}

compareTo vergelijkt 2 cijfers van 2 verschillende studenten met elkaar.

Als het een 10 heeft of tegen komt wordt het tijdelijk vervangen met een final String om verwarring tussen 10 en 1 te voorkomen.

@Override

public int compareTo(Student s) {

final String temp = "99";

if(s.getCijfer().equals("10"))

return cijfer.compareTo(temp);

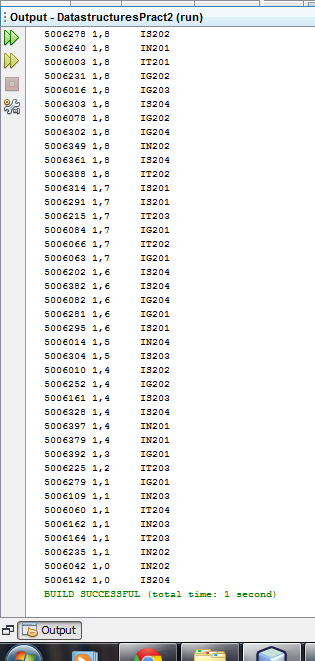
if(this.cijfer.equals("10"))

return temp.compareTo(s.getCijfer());

return cijfer.compareTo(s.getCijfer());

}

Insertion sort Resultaat:



## Test Efficiency

# De SortBucket class

Voor de sortBucket class is ervan uitgegaan dat je een studenten lijst moet interpreteren en niet eerdere objecten die gebruikt zijn bij het genereren van klassen mag gebruiken. De sort methode krijgt een ArrayList met studenten mee.

De eerste stap is om de verschillende klassen uit de studentenlijst te halen. Zodat ze als buckets gebruikt kunnen worden. Daarvoor hebben we eerst de klassennaam in een sorted set gezet. Hierdoor krijg je een lijst met alle unieke klassen die in de studentenlijst voorkomen.

// first find out what classes are in the list so you can make them into buckets

SortedSet<String> klassenSet = new TreeSet();

ArrayList<Klas> klasBucket = new ArrayList<>();

for (int i = 1; i < s.size(); i++) {

String nextKlas = s.get(i).getKlasNummer();

if (!klassenSet.contains(nextKlas)) {

klassenSet.add(nextKlas);

}

}

Daarna worden ze overgezet naar een ArrayList die uit klassen objecten bestaat

//place sorted klassen in arraylist from sortedSet

Iterator it = klassenSet.iterator();

while (it.hasNext()) {

String k = it.next().toString();

Klas temp = new Klas(k);

klasBucket.add(temp);

}

Vervolgens worden de klassen gevuld met studenten.

//place students in klasBucket

for (Student x : s) {

for (Klas k : klasBucket) {

if (k.getClassId().equalsIgnoreCase(x.getKlasNummer())) {

k.addStudent(x);

break;

}

}

}

Nu kunnen we elke bucket sorteren.

//clear the studentList, get the students of each klas in an arrayList and sort on studentNumber, add the contents of the small list to the studentList

s.clear();

for (Klas k : klasBucket) {

ArrayList<Student> sList = k.getStudentList();

for (int i = 0; i < k.getSize(); i++) {

for (int x = i; x > 0 && less(sList.get(x).getStudentNummer(), sList.get(x - 1).getStudentNummer()); x--) {

exch(sList, x, x - 1);

}

}

s.addAll(sList);

}

Met de less methode worden de studenten vergeleken en wordt true gereturned als x is kleiner dan x-1.

private static boolean less(Comparable a, Comparable b) {

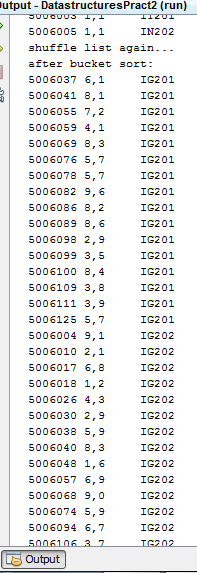
return a.compareTo(b) < 0;

}

private static void exch(ArrayList<Student> s, int x, int y) {

Collections.swap(s, x, y);

}



De output van bucketsort(output van KlasStudentSort is hetzelfde)

# SortKlasStudent

Met de class SortKlasStudent wordt een alternatieve manier uitgevoerd om op klas en student te sorteren. Het is bijna hetzelfde alleen worden geen buckets gebruikt.

public class SortKlasStudent {

public static void sort(ArrayList<Student> s) {

for (int i = 1; i < s.size(); i++) {

for (int x = i; x > 0 && less(s.get(x), s.get(x - 1)); x--) {

exch(s, x, x - 1);

}

}

}

private static boolean less(Student s1, Student s2) {

CompareStudents cs = new CompareStudents();

return cs.compare(s1, s2) < 0;

}

private static void exch(ArrayList<Student> s, int x, int y) {

Collections.swap(s, x, y);

}

}

public class CompareStudents implements Comparator<Student> {

@Override

public int compare(Student s1, Student s2) {

Comparable c1 = s1.getKlasNummer();

Comparable c2 = s2.getKlasNummer();

int klasResult = c1.compareTo(c2);

if (klasResult != 0) {

return klasResult;

}

c1 = s1.getStudentNummer();

c2 = s2.getStudentNummer();

return c1.compareTo(c2);

}

}

# Opdracht 2 b

Voor het testen van de efficiency gebruiken we een array met de waarden 200, 400, 800, 1600, 3200. Voor elke waarde worden verschillende sorteer methoden aangeroepen. En met de stopwatch wordt de verstreken tijd weergegeven. De lijst wordt geshuffled na het sorteren zodat de volgende methode opnieuw kan sorteren.

Stopwatch stopwatch = new Stopwatch();

double t0;

double t1;

double lapTime;

Generator gen;

int[] inputValues = {200, 400, 800, 1600, 3200};

for (int j = 0; j < inputValues.length; j++) {

gen = new Generator(inputValues[j]);

System.out.println(inputValues[j]);

System.out.println("after cijfer sort:");

t0 = stopwatch.elapsedTime();

sortCijfer.Insertion(gen.getStudentList());

t1 = stopwatch.elapsedTime();

lapTime = (1000 \* t1 - 1000 \* t0) / 1000;

System.out.println("(" + t0 + "\t " + t1 + "\t laptime:" + lapTime + ")");

Collections.shuffle(gen.getStudentList());

System.out.println("after KlasStudent sort:");

t0 = stopwatch.elapsedTime();

SortKlasStudent.sort(gen.getStudentList());

t1 = stopwatch.elapsedTime();

lapTime = (1000 \* t1 - 1000 \* t0) / 1000;

System.out.println("(" + t0 + "\t " + t1 + "\t laptime:" + lapTime + ")");

Collections.shuffle(gen.getStudentList());

System.out.println("after bucket sort:");

t0 = stopwatch.elapsedTime();

SortBucket.sort(gen.getStudentList());

t1 = stopwatch.elapsedTime();

lapTime = (1000 \* t1 - 1000 \* t0) / 1000;

System.out.println("(" + t0 + "\t " + t1 + "\t laptime:" + lapTime + ")");

System.out.println("\n");

}

run:

200

after cijfer sort:

(0.04 0.044 laptime:0.004)

after KlasStudent sort:

(0.045 0.051 laptime:0.006)

after bucket sort:

(0.051 0.057 laptime:0.006)

400

after cijfer sort:

(0.08 0.088 laptime:0.008)

after KlasStudent sort:

(0.088 0.093 laptime:0.005)

after bucket sort:

(0.094 0.096 laptime:0.002)

800

after cijfer sort:

(0.123 0.144 laptime:0.021)

after KlasStudent sort:

(0.144 0.156 laptime:0.012)

after bucket sort:

(0.156 0.162 laptime:0.006)

1600

after cijfer sort:

(0.183 0.193 laptime:0.01)

after KlasStudent sort:

(0.194 0.21 laptime:0.016)

after bucket sort:

(0.21 0.228 laptime:0.018)

3200

after cijfer sort:

(0.253 0.286 laptime:0.033)

after KlasStudent sort:

(0.287 0.338 laptime:0.051)

after bucket sort:

(0.338 0.356 laptime:0.018)

# Conclusie

De laptime waarden lopen niet altijd op, misschien komt dit door het shufflen van de lijst waardoor het sorteer algoritme minder werk hoeft te doen of dat de processor in een hogere versnelling gaat werken.

De resultaten zijn anders dan de theorie want de sortBucket class heeft O(N^6):

For loop  
while loop  
for loop  
for loop{for loop{ for loop}}

En Insertion sort heeft O(N^2):

For loop

While loop

Hij moet meerdere keren de hele studentenlijst door zoeken. Toch is deze in de resultaten het snelste.