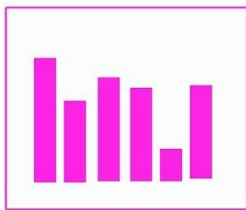


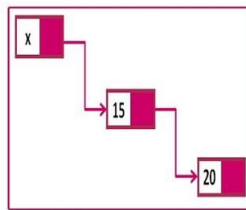
# Practicum 2 Datastructures

*Yacob Ben Youb en Youssef Ben Youb*

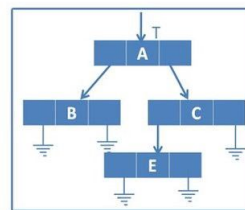
500672040 - 500775494



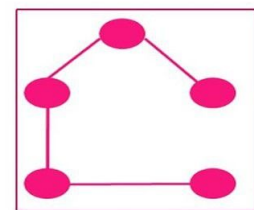
Sorting



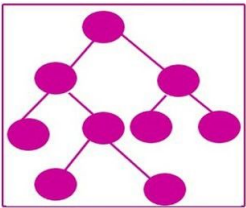
Link list



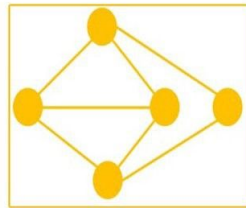
list



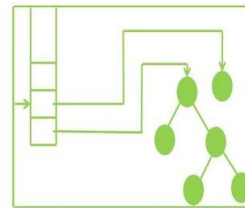
spanning tree



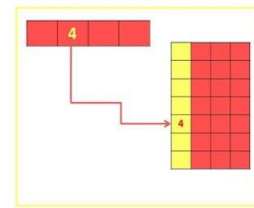
Tree



Graph



Stack



Hashing

<b>Klassen:</b>	<b>2</b>
<b>Approach 1: Sorting after adding to list</b>	<b>3</b>
<b>Variant 1: using Selection Sort</b>	<b>3</b>
<b>Variant 2: using Bucket Sort</b>	<b>5</b>
<b>Approach 2: Priority Queue</b>	<b>8</b>

## Klassen:

Alle klassen behalve Player zijn een implementatie van HighScoreList. Deze implementaties zijn verschillende methodes om op een gesorteerde methodes de highscores terug te geven.

Er zijn hier drie verschillende implementaties van (exclusief NormalSort, die hoort niet bij het practicum).

Selectionsort, Bucketsort en Priorityqueue. Deze klassen worden allemaal getest in HighScoreListTest. Deze klasse is verbeterd en er zijn extra tests aan toegevoegd om de efficiëntie van de sorteer algoritmes te testen. Het argument -Xint wordt gebruikt om de tests gecontroleerd te laten verlopen.

## Approach 1: Sorting after adding to list

In deze opdracht gaan we sorteren door de elementen in het begin van de lijst toe te voegen en deze hierna te verschuiven naar de juiste plek.

### Variant 1: using Selection Sort

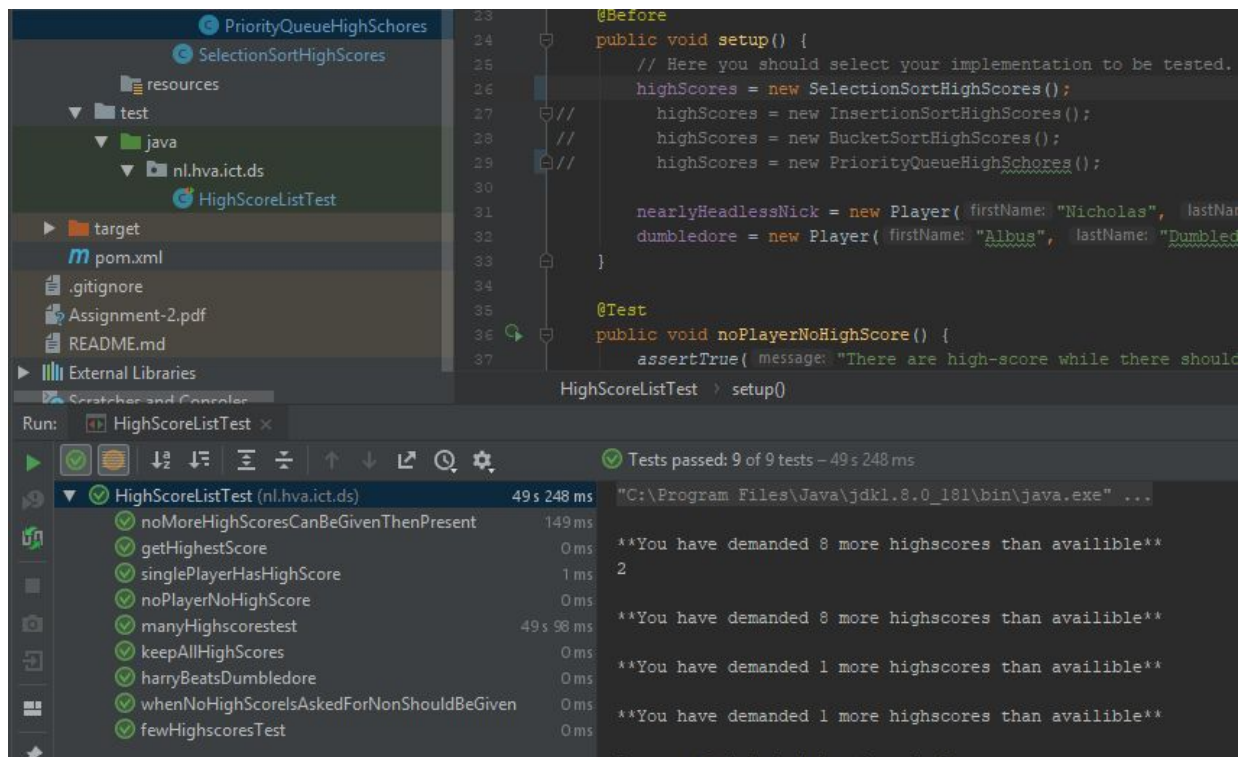
Selection sort add een player aan de lijst en kijkt vervolgens of hij een groter element in de lijst kan vinden.

```
public void add(Player player) {  
    //voeg een player toe aan het einde van de lijst  
    players.add(player);  
    players = selectionSort(players);  
}
```

Zo ja dan kijkt hij of hij een grotere element dan dat element kan vinden. Dit gaat door totdat het einde van de lijst bereikt wordt. Het grootste gevonden element wordt daarna met het eerste element omgewisseld.

```
public List<Player> selectionSort(List<Player> players){  
    List<Player> tempList = players;  
    //als de lijst slechts een player bevat hoeft je niet te sorteren dus ben je  
    klaar  
    if (tempList.size() <= 1) {  
        return tempList;  
    }  
    //loopt van begin tot de totale grootte van de lijst  
    for (int i = 0; i < tempList.size()-1; i++) {  
        //zet de tijdelijke indexen gelijk aan de begin index  
        int loopIndex = i;  
        int holderIndex = i;  
        //zolang de huidige vergelijk plaats kleiner is dan de maximum grootte min  
        een  
        while ((loopIndex) < tempList.size()-1) {  
            //vergelijk de tijdelijke index met de volgende index  
            if (tempList.get(holderIndex).getHighScore() <  
tempList.get(loopIndex+1).getHighScore()) {  
                //als de tijdelijke index kleiner is dan de volgende index is de  
                volgende index de tijdelijke index  
                holderIndex = loopIndex+1;  
            }  
            //verhoog de huidige loop index met 1 zodat de loop conditie de volgende  
            index checkt  
            loopIndex++;  
        }  
        //als er een grotere waarde dan de eerste index is gevonden wordt de  
        grootste waarde verwisseld met de eerste  
        if (holderIndex > i){  
            Player temp = tempList.get(holderIndex);  
            tempList.set(holderIndex, tempList.get(i));  
            tempList.set(i, temp);  
        }  
    }  
    //geef de gesorteerde lijst terug.  
    return tempList;  
}
```

Voor het sorteren van 1000 players met selectionsort kost dit ongeveer 50 seconden. (Zie manyHighScoresTest in de lijst)



The screenshot shows an IDE with a project structure on the left and a code editor on the right. The project structure includes a package `nl.hva.ict.ds` with a class `HighScoreListTest`. The code editor shows the `HighScoreListTest` class with a `@Before` method `setup()` and a `@Test` method `noPlayerNoHighScore()`. The `setup()` method initializes `highScores` with `SelectionSortHighScores()` and creates `nearlyHeadlessNick` and `dumbledore` players. The `noPlayerNoHighScore()` method asserts that there are high-scores while there should be none. The bottom of the screenshot shows the test results for `HighScoreListTest`, indicating that 9 out of 9 tests passed in 49s 248ms. The tests include `noMoreHighScoresCanBeGivenThenPresent`, `getHighestScore`, `singlePlayerHasHighScore`, `noPlayerNoHighScore`, `manyHighscorestest`, `keepAllHighScores`, `harryBeatsDumbledore`, `whenNoHighScoresAskedForNonShouldBeGiven`, and `fewHighscoresTest`.

De Big O van selectionsort is:  $O(n^2)$ . Elke extra additie kan het algoritme exponentieel langer laten verlopen.

## Variant 2: using Bucket Sort

Bucket sort is een implementatie van Selection sort maar dan over meerdere lijsten. Elke lijst loopt van een bepaald nummer tot een ander bepaald nummer. Voor onze test hebben we 5 lijsten aangemaakt

```
//Maakt 5 bucketlists aan
public BucketSortHighScores() {
    bucketList.add(new ArrayList<>());
    bucketList.add(new ArrayList<>());
    bucketList.add(new ArrayList<>());
    bucketList.add(new ArrayList<>());
    bucketList.add(new ArrayList<>());
}

@Override
//Voegt players toe in bijbehorende bucket en voert daarna selectionsort uit op
de bucket waarin een player is toegevoegd.
public void add(Player player) {
    //Als score hoger is dan 80.000, voeg de player toe in deze bucket. Voor de andere
    buckets wordt vergeleken met 60k, 40k en 20k en daarna automatisch in de laatste
    bucket
    // zodat negatieve scores ook ingevoerd kunnen worden.
    if (player.getHighScore() > bucket1MinScore) {
        bucketList.get(0).add(player);
        bucketList.set(0, selectionSort.selectionSort(bucketList.get(0)));
    }
}
```

```
//          printAllBucketHighScores();
        return;
    }
    if (player.getHighScore() > bucket2MinScore) {
        bucketList.get(1).add(player);
        bucketList.set(1, selectionSort.selectionSort(bucketList.get(1)));
    }
    printAllBucketHighScores();
    Return;
//etc etc etc
}
```

Omdat de maximum highscore gedefinieerd staat als 100.00 heb ik hier vijf buckets voor gemaakt, elk 20.000 apart.

```
@Override
public List<Player> getHighScores(int numberOfHighScores) {
    //maakt list aan om te returnen
    List<Player> returnList = getAllPlayersFromBucketList(bucketList);
    //als er meer highscores worden opgevraagd dan er zijn
    if (numberOfHighScores > returnList.size()) {
        //print dit uit
        System.out.println("\n**You have demanded " + (numberOfHighScores -
returnList.size()) + " more highscores than available**");
        //en voer de functie daarna uit met het maximale aantal highscores
mogelijk
        return getHighScores(returnList.size());
    }
    //Omdat het uigesloten is dat numberOfHighscores nu nog groter is dan
Players kan dit weggelaten worden.
    List<Player> returnHighscores = new ArrayList(returnList.subList(0,
numberOfHighScores));
    //          System.out.println("GEVRAAGDE HIGHSCORES");
    //          for (Player plaay : returnHighscores) {
    //              System.out.println(plaay.getHighScore());
    //          }
    return returnHighscores;
}
```

Om de bucketlist terug te krijgen moeten we stuk voor stuk door elke lijst heen lopen en alle players op volgorde eruit halen en samenvoegen in een gezamenlijke lijst.

```
//Haalt alle players stuk voor stuk uit elke bucketlist.
public List<Player> getAllPlayersFromBucketList(List<List<Player>> bucketList) {
    //Maak een arraylist aan waarin alle elementen bij elkaar gevoegd kunnen
worden.
    List<Player> returnList = new ArrayList<>();
    //Voor elke bucket in de bucketlist
    for (List<Player> playerList : bucketList) {
        //Voor elke player in de bucket
        for (Player player : playerList) {
            //Voeg de player toe aan de returnlist.
            returnList.add(player);
        }
        //          System.out.println(player.getHighScore());
    }
    return returnList;
}
```

*Bucket sort met een grootte van 1000 players. In plaats van 49 seconden kost dit nu 2 seconden.*

The screenshot shows an IDE with a Java project. The code in the editor defines a `BucketSortHighScores` class and a `HighScoreListTest` class. The test class includes a `noPlayerNoHighScore` test method. The test results pane at the bottom shows that all tests passed, with a total execution time of 2 s 318 ms. The test results are as follows:

Test Name	Duration	Status
HighScoreListTest (nl.hva.ict.ds)	2 s 318 ms	Passed
noMoreHighScoresCanBeGivenThenPresent	161 ms	Passed
getHighestScore	0 ms	Passed
singlePlayerHasHighScore	0 ms	Passed
noPlayerNoHighScore	0 ms	Passed
manyHighscorestest	2 s 157 ms	Passed
keepAllHighScores	0 ms	Passed
harryBeatsDumbledore	0 ms	Passed
whenNoHighScoresAskedForNonShouldBeGiven	0 ms	Passed
fewHighscoresTest	0 ms	Passed

Dit betekent dat het ongeveer even snel zou moeten runnen als vijf Selectionsorts van  $\frac{1}{5}$  grootte. Oftewel  $5 \cdot (O(5))^2$ . Laten we kijken of dit klopt. We vermenigvuldigen het aantal spelers met 5 om dit te testen.

*Bucket sort met 5000 players kost 4min en 3 sec*

The screenshot shows the test results for the `manyHighscorestest`. The test passed with a duration of 4 m 3 s 5 ms.

Test Name	Duration	Status
manyHighscorestest	4 m 3 s 5 ms	Passed

Het kost nu 4 en 3 seconden minuten in plaats van 49 seconden. Vijf keer 49 seconden = 250 seconden = 4 minuten en 5 seconden. Dit komt ongeveer overeen met onze initiele inschatting omdat het sorteren soms langer of minder lang kost op basis van of de nieuwe nummers aan het begin of einde van de lijst moeten komen, is het verschil van 2 seconden dus binnen de perken van onze verwachtingen. Om het bucket sorten nog sneller te maken zou er gekeken kunnen worden naar het totaal aantal players en meerdere buckets maken op basis van de minimum en maximum score. 5 buckets is te weinig voor 5000 spelers om efficiënt te sorteren.

Indien deze factoren van tevoren bekend zouden zijn zou de efficiëntie verbeterd kunnen worden tot  $\Theta(n+k)$  voor bucket sort.

## Approach 2: Priority Queue

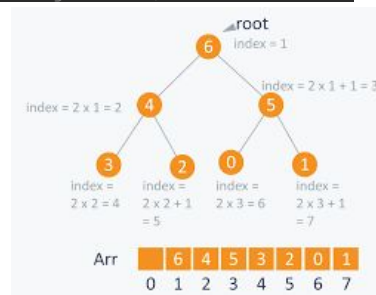
Een priority queue volgt een heap structuur. Elk object heeft twee sub objecten welke groter of kleiner zijn dan dat object. Om dit te bepalen moet voor niet-standaard objecten een eigen vergelijker (comparator) worden gemaakt.

```
/*Custom comparator class, deze wordt gebruikt voor de priorityqueue om te beslissen of een player groter of kleiner is dan de volgende. Dit bepaald hij op basis van de player highscores
```

```
Wat belangrijk is, is dat dit een descending queue is. Gewoonlijk komt het kleinste nummer eerst en het grootste nummer laatst, maar onze comparator draait dit om.*/  
class PlayerComparator implements Comparator<Player> {
```

Indien dit niet gedaan wordt zal er door java zelf een vergelijk methode worden gekozen. Deze zal niet altijd zijn waarop men wil sorteren. Wanneer er een object toegevoegd wordt, wordt deze met het eerste object vergeleken. Als hij kleiner is springt de vergelijker naar het object linksonder, als hij groter is naar het object rechtsonder. Hierna wordt er opnieuw vergeleken. Op deze manier hoeft er veel minder vergeleken worden dan bij een selectionsort of bucketsort methode. We laten onze comparator op de Player's highscore vergelijken.

```
//Wanneer compare wordt gebruikt moeten er twee players worden meegegeven  
public int compare(Player x, Player y) {  
    //als de eerdere highscore groter is dan de tweede highscore, doe niks  
    if (x.getHighScore() > y.getHighScore()) {  
        return -1;  
    }  
    //Als de eerdere highscore kleiner is dan de tweede highscore, wissel ze om  
    if (x.getHighScore() < y.getHighScore()) {  
        return 1;  
    }  
    //Als de highscores hetzelfde zijn, doe niks.  
    return 0;  
}
```



Eerst wordt een object toegevoegd aan de priority queue

```
Comparator comparator = new PlayerComparator() ;  
//Maakt een priorityqueue en gebruikt de playerComparator om te sorteren op  
highscore basis  
PriorityQueue<Player> priorityQueue = new PriorityQueue<>(comparator);  
@Override  
Voeg een player toe aan de priority queue. De add methode gebruikt de comparator om uit  
te vinden waar het in de queue ge add moet worden.*/  
public void add(Player player) {  
    priorityQueue.add(player);  
}
```

Ook is het belangrijk om de objecten uit de priorityqueue te kunnen halen. Om dit te doen wordt er door de queue heen geloopt en worden alle elementen stuk voor stuk in een ArrayList gestopt.

```
@Override
public List<Player> getHighScores(int numberOfHighScores) {
    //als er meer highscores worden opgevraagd dan er zijn
    if (numberOfHighScores > priorityQueue.size()) {
        //print hoeveel highscores er teveel zijn opgevraagd
        System.out.println("\n**You have demanded " + (numberOfHighScores -
priorityQueue.size()) + " more highscores than available**");
        //en voer de functie daarna uit met het maximale aantal highscores mogelijk
        return getAllHighScores();
    }
    List<Player> returnList = new ArrayList<>();
    //Haal alle players op uit de priorityqueue en zet ze in een player List
    int q = 0;
    for (Player play : priorityQueue){
        //onnodige optimalisatie
        if (q<=numberOfHighScores){
            returnList.add(play);
        }
        else {break;}
    }
    //System.out.println(play.getHighScore());
    //Omdat het uigesloten is dat numberOfHighscores nu nog groter is dan
    Players kan dit weggelaten worden.
    return returnList.subList(0, numberOfHighScores);
}
```

Priority queue werkt vele malen efficiënter dan de andere twee sorteermethodes. Dit komt omdat het bij het invoegen van het element al de goede plaats selecteert en er niet achteraf gesorteerd hoeft te worden. Het hoeft slechts een beperkt aantal stappen hoeft te doorlopen om de plaats te vinden waar het element geplaatst moet worden

*Zoals hieronder te zien valt, kost het slechts 8 milliseconden om een lijst van 1000 elementen gesorteerd terug te geven voor een priority queue.*

The screenshot shows an IDE with a project named 'nl.hva.ict.ds'. The 'HighScoreListTest' class is selected in the left sidebar. The right pane shows the source code of 'HighScoreListTest' with the following content:

```
highScores = new PriorityQueueHighScores();
nearlyHeadlessNick = new Player( firstName: "Nick",
dumbledore = new Player( firstName: "Albus", last
}
@Test
public void noPlayerNoHighScore() {
    assertTrue( message: "There are high-score while"
}
```

The bottom pane shows the test results for 'HighScoreListTest (nl.hva.ict.ds)'. The tests passed are:

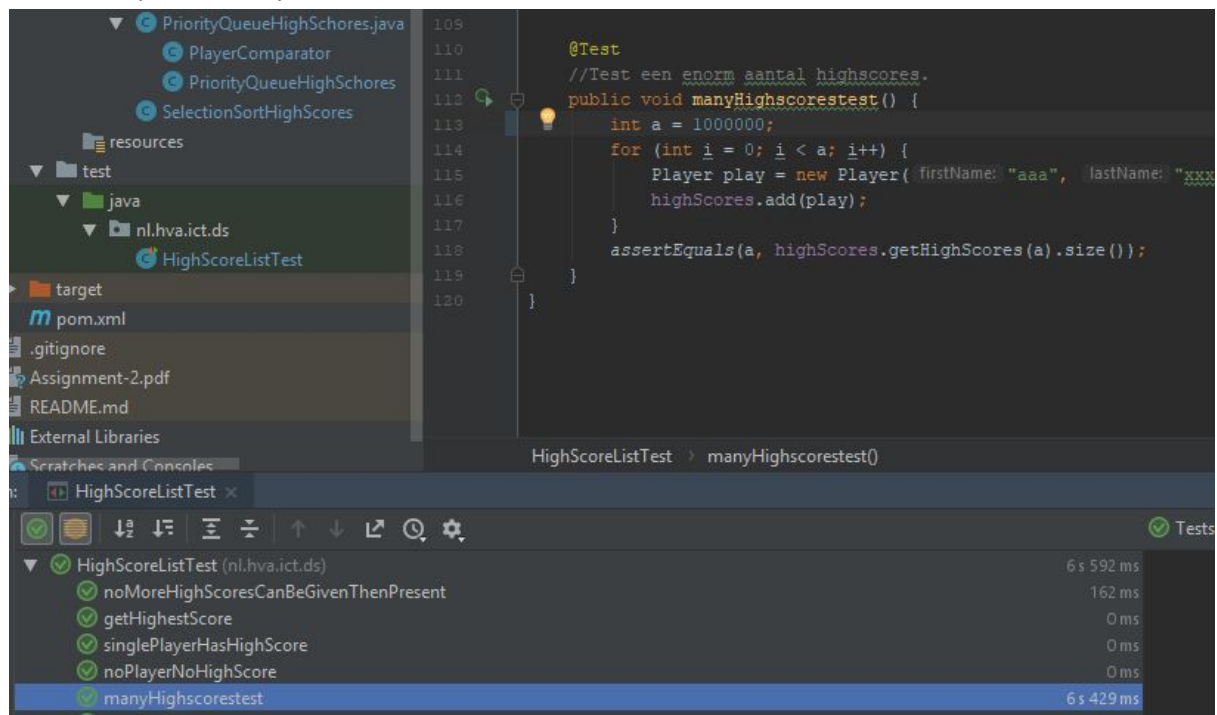
Test Name	Duration
HighScoreListTest (nl.hva.ict.ds)	160 ms
noMoreHighScoresCanBeGivenThenPresent	151 ms
getHighestScore	0 ms
singlePlayerHasHighScore	1 ms
noPlayerNoHighScore	0 ms
manyHighscorestest	8 ms
keepAllHighScores	0 ms
harryBeatsDumbledore	0 ms
whenNoHighScoresAskedForNonShouldBeGiven	0 ms
fewHighscoresTest	0 ms

The status bar at the bottom indicates 'Tests passed: 9 of 9 tests - 160 ms'.



Pas bij extreem hoge getallen begint de priority queue enige tekenen van vertraging te laten zien

1.000.000 ipv 1000 Players: 6 seconden.



Zelfs met duizend maal zoveel spelers is de priority queue sneller dan een selection sort gebaseerd algoritme. De big O van een priority queue is namelijk  **$O(\log(n))$**

## Conclusie:

Omdat er veel variatie is tussen de highscores en er veel spelers kunnen zijn, is de **priority queue** duidelijk de beste optie. Indien het aantal spelers boven de honderdtallen groeit laat de priority queue al snel blijken dat hij met kop en schouders de snelste en efficiëntste is.