ALGORITMES & DATASTUCTUREN

Joost Visser (jwjh.visser@avans.nl)

Academie voor Deeltijd

2 juni 2021

Opnemen

Deze lecture wordt opgenomen. Vinden jullie dit goed?

Les van vandaag

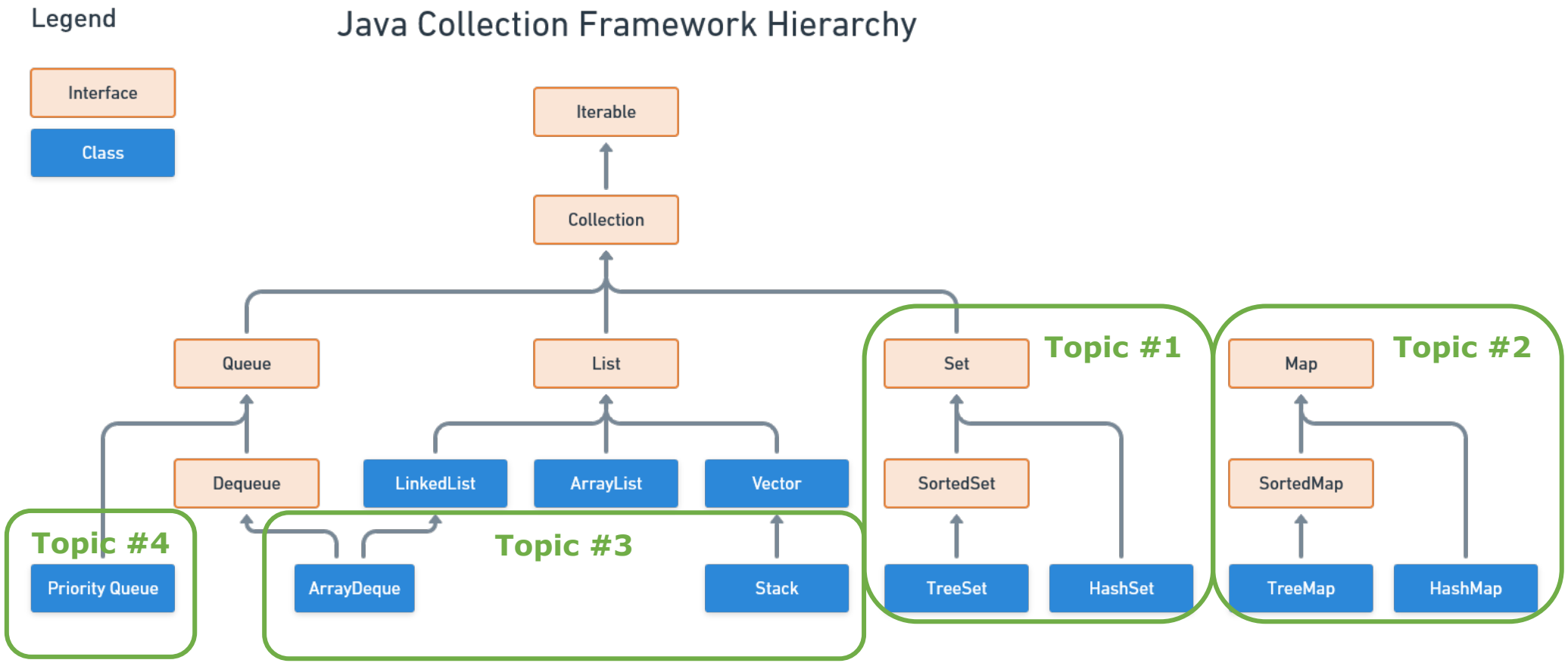
Lesplan

Les	Datum	Topic	Inhoud
1	4 mei 2021	Introductie	Inleiding, algoritme complexiteit, data-structuren, Linear zoeken, binair zoeken
2	11 mei 2021	Datastructuren 1	Herhaling algoritme complexiteit, Sorteren 1, Abstracte datatypen, Array, List, Set, Hashing
3	18 mei 2021	Datastructuren 2	**Sets 2**, **Maps,** Stack, Deque, Heap, Priority Queue,
4	25 mei 2021	Algoritmes	**Generieke datatypen**, **(**Sorteren 2**)**, recursie, binaire bomen, proeftoets

Vragen?

Les van vandaag

Java datastructuren



Les van vandaag

Inhoudsopgave

1. **Opdrachten van vorige week**
    1. Welke opdrachten willen jullie bespreken?
2. Set 2
3. Maps
4. ArrayDeque
5. PriorityQueue
6. Generics

1. Opdrachten van vorige week

Welke opdrachten willen jullie bespreken?

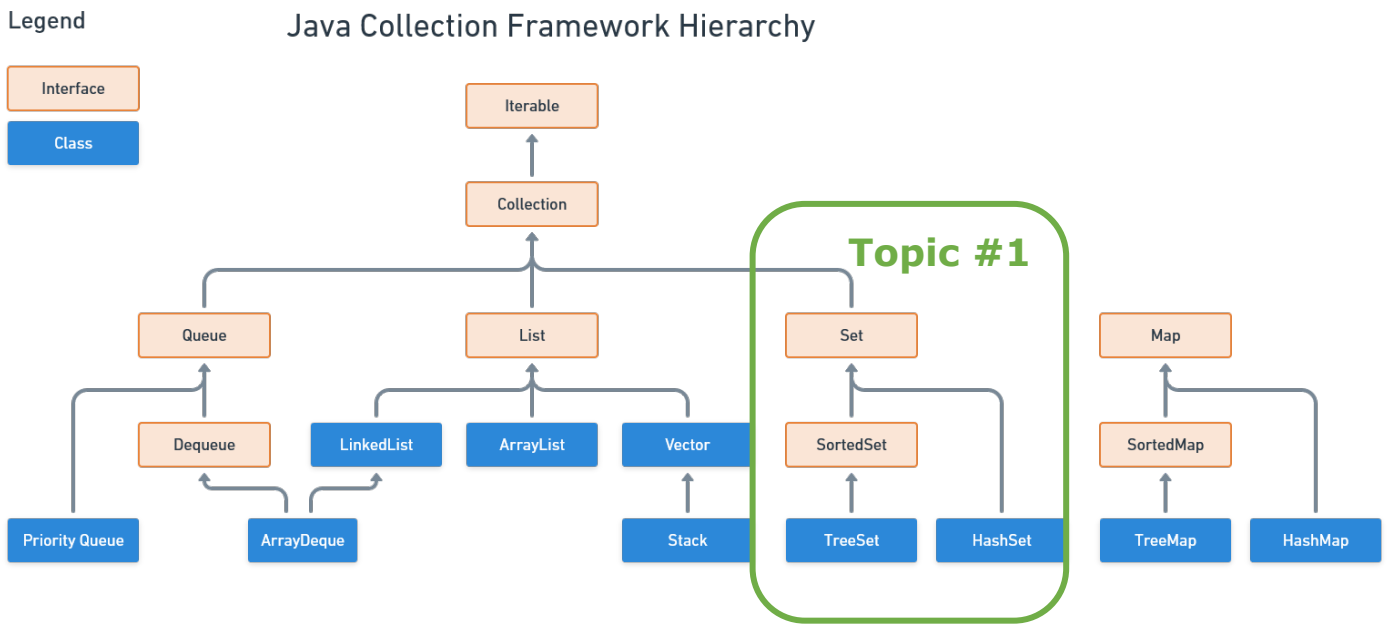
Dit waren de opdrachten van vorige week:

1. Sorting:
    1. Implementeer een custom sorting algoritme met Vogelbekdieren
2. List en ArrayList:
    1. [Toets] Implementeer VogelbekdierSorting met ArrayList
    2. Implementeer custom LinkedList
3. HashSet en TreeSet:
    1. [Toets] Implementeer $O(n \log n)$ sorting met vogelbekdieren en TreeSets.
    2. [Toets] Implementeer een HashSet met vogelbekdieren.

Les van vandaag

Inhoudsopgave

1. Opdrachten van vorige week
2. **Set 2**
   1. Herhaling sets
   2. Custom classes: compareTo, equals en hashCode()
3. Maps
4. ArrayDeque
5. PriorityQueue
6. Generics



2. Set 2

Herhaling Sets

1. Een set is een verzameling van objecten.
   Het is vergelijkbaar met de volgende verschillen:
   1. Duplicates worden niet bewaard
   2. Je kan niet objecten op index aanroepen (i.e. `set.get(3)` werkt niet)

Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**

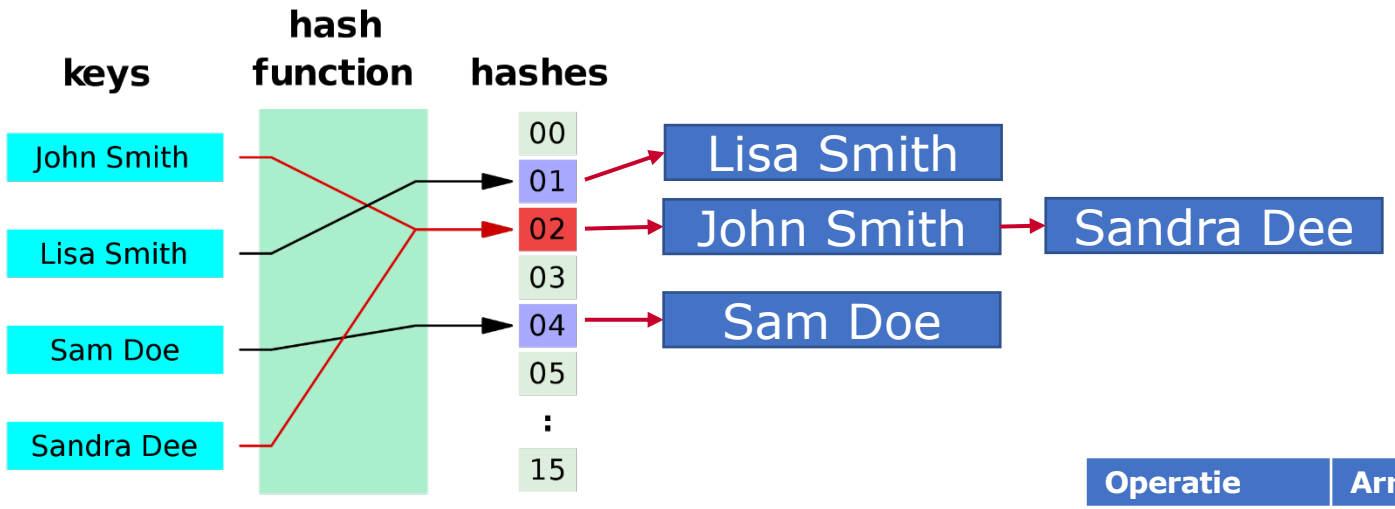
*Tenzij de rij vol zit, dan is het één keer $O(n)$. Gemiddeld worst-case nog steeds $O(1)$, dit noemen we de *amortized running time*.

**Ligt aan het aantal collisions van de Hash, maar gemiddeld $O(1)$. Sinds Java 8 is de *worst-case* $O(\log n)$, omdat ze de hash list van een LinkedList naar een TreeSet veranderen bij te veel collisions.

2. Set 2

Herhaling Sets

HashSet: sla objecten op doormiddel van hun HashCode.



Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**

2. Set 2

Herhaling Sets

TreeSet: sla objecten op in een Red-Black tree (trees → Les 4).

Voordelen van TreeSet t.o.v. HashSet:

1. Elementen zijn gelijk gesorteerd.
2. Kan éérste en laatste element pollen $O(\log n)$ + paar andere extra functies

Nadeel:

1. $O(\log n)$ in plaats van $O(1)$

Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**

2. Set 2

Custom classes: compareTo, equals en hashCode()

Wil je een HashSet met custom classes?

1. Implementeer equals() en **hashCode()**

```
@Override
public int hashCode() {
    return Objects.hash(this.property1, this.property2, …);
}
```

- Return de hashCode int

Dit implementeert een snelle wiskundige formule met priemgetallen wat vaak unieke hashes geeft.

Soms doet je IDE dit automatisch.

```
@Override
public final int hashCode() {
    int result = 17;
    result = 31 * result + city.hashCode();
    result = 31 * result + department.hashCode();
    return result;
}
```

2. Set 2

Custom classes: compareTo, equals en hashCode()

Wil je een HashSet met custom classes?

1. Implementeer **equals()** en hashCode()

```
public boolean equals(Object obj)
```

- Return True als *obj* hetzelfde is als *this*
- Return False anders

```
@Override
public boolean equals(Object obj) {
    if (! (obj instanceof Vogelbekdier))  return false;
    if (getClass() != obj.getClass()) return false;

    Vogelbekdier andereDier = (Vogelbekdier) obj;
    return naam.equals(andereDier.naam) && lengte == andereDier.lengte;
}
```

2. Set 2

Custom classes: compareTo, equals en hashCode()

Wil je een TreeSet met custom classes?

1. Implementeer equals() en **compareTo()**

Stap 1: zorg ervoor dat de classe de Comparable interface implementeert.

```
public class Vogelbekdier implements Comparable<Vogelbekdier>
```

Stap 2: Implementeer de compareTo functie, dit bepaalt de ordering.

1. Return een negatief getal als huidige object *this* < Object o
2. Return 0 als huidige object *this* = Object o
3. Return een positief getal als huidige object *this* > Object o

```
public int compareTo(Vogelbekdier o)
```

2. Set 2

Custom classes: compareTo, equals en hashCode()

Wil je een TreeSet met custom classes?

1. Implementeer equals() en **compareTo()**

Stap 2: Implementeer de compareTo functie, dit bepaalt de ordering.

Sorteer op integers:

```
@Override
public int compareTo(Vogelbekdier o) {
    return lengte - o.lengte;
}
```

Alternatief:

```
@Override
public int compareTo(Vogelbekdier o) {
    return Integer.compare(lengte, o.lengte);
}
```

Sorteer op Strings:

```
@Override
public int compareTo(Vogelbekdier2 o) {
    return naam.compareTo(o.naam);
}
```

2. Set 2

Custom classes: compareTo, equals en hashCode()

Wil je een TreeSet met custom classes?

1. Implementeer equals() en **compareTo()**

In plaats van de `compareTo()` te implementeren, kan je ook een Comparator doorgeven bij het aanmaken van de TestSet.

```
TreeSet<Docent> treeSet = new TreeSet<>((e1, e2) -> e1.naam.compareTo(e2.naam));
```

Voordelen:

1. Minder code te schrijven
2. Verschillende sorteerfuncties mogelijk.

Nadeel:

1. Mocht je meerdere keren een bepaald object in een set moeten gebruiken, dan moet je deze comparator steeds copy-pasten.
2. Je moet hetgene wat je wilt comparen public maken.)

2. Set 2

Samenvatting

Een set is een verzameling van objecten, zonder duplicates die *niet direct* op index aangeroepen kunnen worden.

Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**

2. Set 2

Samenvatting

Een set is een verzameling van objecten, zonder duplicates die niet direct op index aangeroepen kunnen worden.

TreeSet: Een set met $O(\log(n))$ operaties die automatisch de set sorteerd.

HashSet: Een set met $O(1)$ operaties die gebruik maakt van hashes.

HashSet of **TreeSet** gebruik met eigen classes?

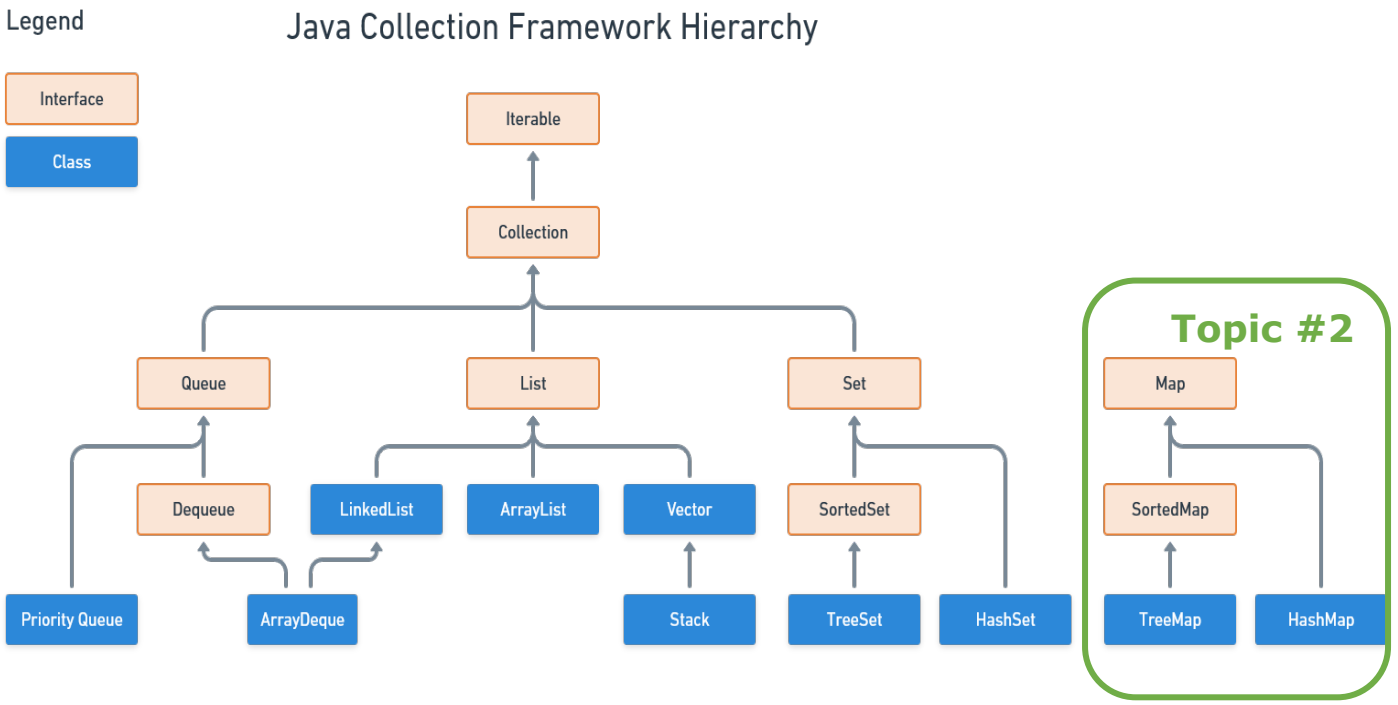
1. HashSet: Implementeer `equals()` en `hashCode()`
2. TreeSet: Implementeer `equals()` en `compareTo()` (waarbij je voor de laatste de interface `Comparable<>` moet implementeren).

Vragen?

Les van vandaag

Inhoudsopgave

1. Opdrachten van vorige week
2. Set 2
3. **Maps**
   1. Wat zijn maps?
   2. Hoe gebruik je maps?
4. ArrayDeque
5. PriorityQueue
6. Generics



3. Maps

Wat zijn maps?

Hebben jullie ooit wel eens een keer iets opgezocht?

1. Gegeven naam → Facebook profiel
2. Gegeven docent → Vakken
3. Studie → Vakken
4. Vak → Studie

In Java kunnen we dit implementeren met Maps.

Met Maps kunnen wij via een key de bijbehorende value opzoeken.

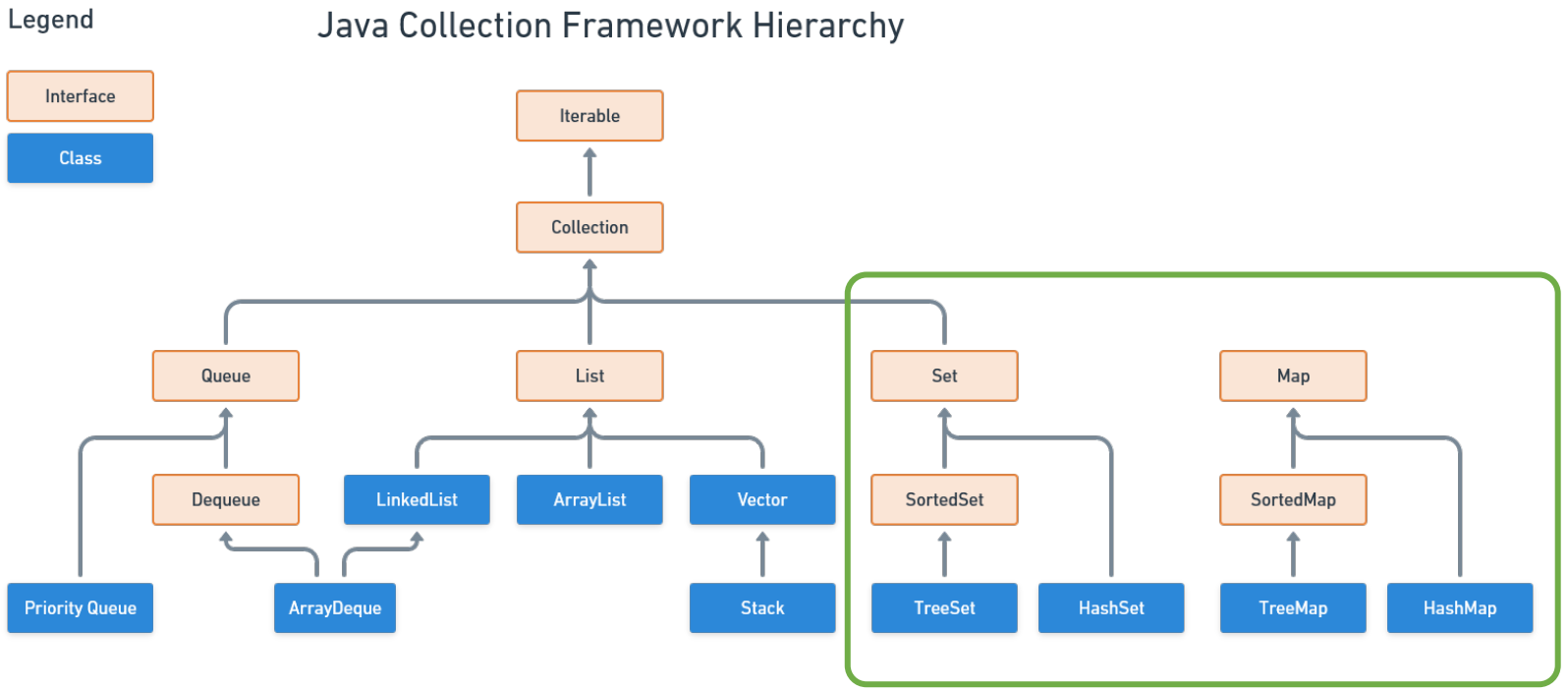
Key → Value(s)

- In Python, Javascript en C#: Dictionary
- In Java: `Map<String, String> myMap;`

3. Maps

Wat zijn maps?

Valt jullie iets op?



3. Maps

Wat zijn maps?

In principe werken maps op eenzelfde manier als sets, met een extra functionaliteit: ze wijzen naar bepaalde data.

- Waarom doen we dit niet met ArrayLists? **Antwoord:** duplicates.

Enorm handig voor het opzoeken van data, gegeven een key.

Er zijn twee implementaties die we bespreken:

1. TreeMap
2. HashMap

Deze werken hetzelfde als TreeSet en HashSet!
Ze hebben ook dezelfde running time.

3. Maps

Hoe gebruik je maps?

Relevante functies:

1. `get(k)` [en `getOrDefault(k, 0)`]
   - Vind values gegeven key
2. `put(k, v)`
   - Insert value v met key k.
3. `remove(k)`
   - Removes de key k en bijbehorende value
4. `keyset()`
   - Een Set met alle key
5. `values()`
   - Een Collection met alle values (waar je met een forEach loop kan iteraten)

3. Maps

Hoe gebruik je maps?

Voorbeeld 1: Naam → Docent

```
public class Docent {
    int id;
    String naam;
    int werkErvaring;

    public Docent(int id, String naam, int werkErvaring) {
        this.id = id;
        this.naam = naam;
        this.werkErvaring = werkErvaring;
    }
}
```

```
public static void main(String[] args) {
    HashMap<String, Docent> naamMapping = new HashMap<>();
    naamMapping.put("Joost",    new Docent(1, "Joost",  1));
    naamMapping.put("Patrick",  new Docent(2, "Patrick", 20));
    naamMapping.put("Frans",    new Docent(3, "Frans", 40));
    System.out.println("Werkervaring Joost: " + naamMapping.get("Joost").werkErvaring);
}
```

System.out: “Werkervaring Joost: 1”

Vraag: Moet ik hiervoor hashCode implementeren in Docent?

3. Maps

Hoe gebruik je maps?

Voorbeeld 2: Naam → Docent

Print docenten in alfabetische volgorde.

```
public class Docent {
    int id;
    String naam;
    int werkErvaring;

    public Docent(int id, String naam, int werkErvaring) {
        this.id = id;
        this.naam = naam;
        this.werkErvaring = werkErvaring;
    }
}
```

```
public static void main(String[] args) {
    TreeMap<String, Docent> naamMapping = new TreeMap<>();
    naamMapping.put("Joost",    new Docent(1, "Joost",  1));
    naamMapping.put("Patrick",  new Docent(2, "Patrick", 20));
    naamMapping.put("Frans",    new Docent(3, "Frans", 40));
    System.out.println("Werkervaring Joost: " + naamMapping.get("Joost").werkErvaring);
    for (Docent docent: naamMapping.values()) {
        System.out.println(docent.naam);
    }
}
```

System.out

Werkervaring Joost: 1
Frans
Joost
Patrick

3. Maps

Hoe gebruik je maps?

Voorbeeld 3: Docent → Vakken

```
public class Docent {
    int id;
    String naam;
    int werkErvaring;

    public Docent(int id, String naam, int werkErvaring) {
        this.id = id;
        this.naam = naam;
        this.werkErvaring = werkErvaring;
    }
}
```

```
public static void main(String[] args) {
    HashMap<String, ArrayList<Vak>> vakMapping = new HashMap<>();
    …
}
```

Deze mogen jullie straks zelf implementeren. ☺

3. Maps

Samenvatting

Maps zijn datastructuren waarmee je makkelijk dingen kan opzoeken.

Key → Value

In Java worden ze als HashMap en TreeMap geïmplementeert, wat erg op de HashSet en TreeSet lijkt (met dezelfde running times).

Gebruik HashMap als je keys/values niet gesorteerd hoeft te hebben, gebruik anders TreeSet.

Vragen?

3. Maps

Opdrachten

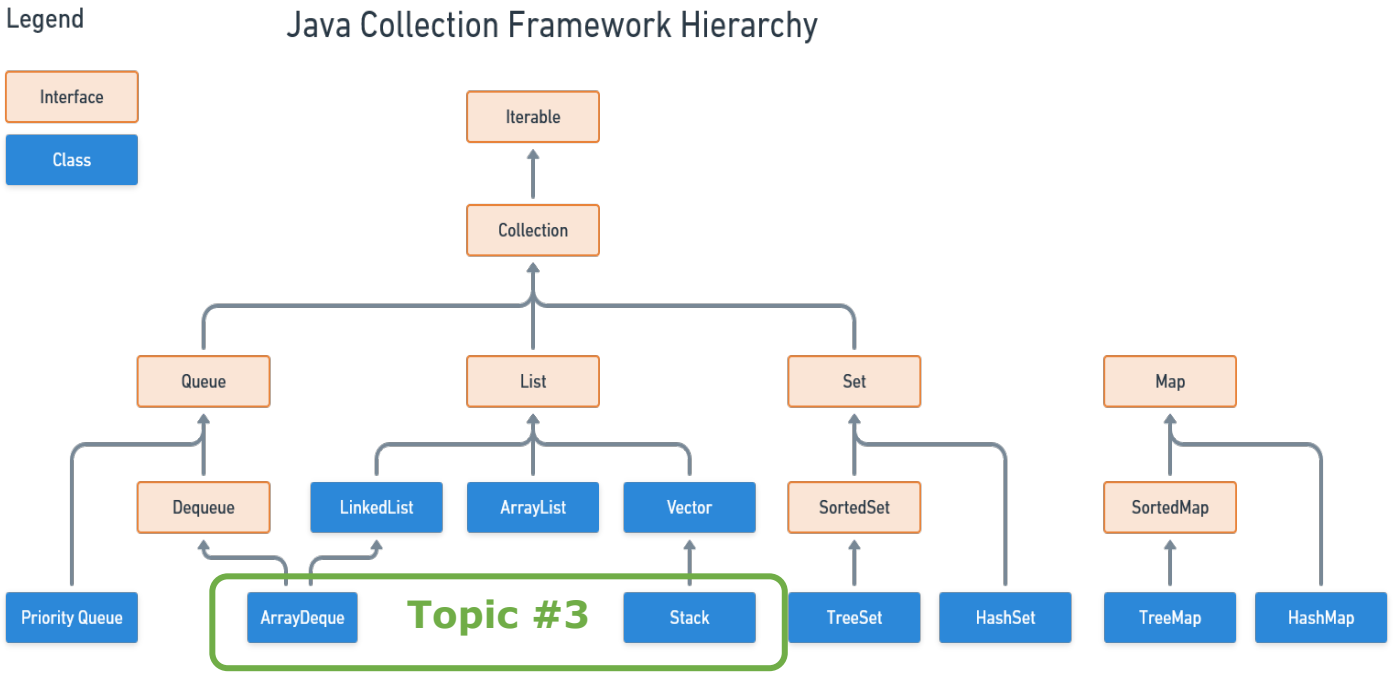
1. Implementeer een mapping van Naam → Vogelbekdier in de vogelbekdieren opdracht. [VogelbekdierMaps.java]
2. In hetzelfde bestand, implementeer “Sorteren van vogelbekdieren” opdracht met TreeMaps. (Sorteer op integer en op naam.) Gebruik géén .sort()
3. Implementeer de mapping van Docent → Vakken. [DocentMaps.java]
   1. Joost → ADS, Wiskunde
   2. Patrick → ADS, Design Patterns

   Verifieer dat je implementatie werkt. (Krijg je de vakken ADS en Wiskunde als je `get(“Joost”);` uitvoert? Bijvoorbeeld.)

Les van vandaag

Inhoudsopgave

1. Opdrachten van vorige week
2. Set 2
3. Maps
4. **ArrayDeque**
    1. Wat zijn queues?
    2. FIFO en LIFO queues
    3. ArrayDeque
5. PriorityQueue
6. Generics



4. ArrayDeque

Wat zijn queues?

Ooit wel eens in een rij gestaan?

Soms is de orde van de data belangrijk, dan kunnen we queues gebruiken.

Een queue is een rij aan objecten met een orde.

“Stefano” → “Luuk” → “Marissa” → “Claire”

Als je dan een extra element aan de queue toevoegt, dan sluit het achteraan. Stel we voegen “Lisa” toe:

“Lisa” → “Stefano” → “Luuk” → “Marissa” → “Claire”

4. ArrayDeque

FIFO en LIFO queues

Nu kunnen we aan twee kanten bij de rij er mensen uithalen:
"Lisa" → "Stefano" → "Luuk" → "Marissa" → "Claire"

Haal je het object eruit wat het laatste erin zit? LIFO Queue:

- LIFO Queue → Last-In-First-Out queue
- Ook wel een stack genoemd.
- **Voorbeelden:** Stapel borden. Undo/Redo stack in Word.

Haal je het object eruit wat het eerste erin zit? FIFO Queue:

- FIFO Queue → First-In-First-Out queue
- Ook wel een queue genoemd.
- **Voorbeelden:** Wachtrij voor een kassa. Printer queue.

4. ArrayDeque

FIFO en LIFO queues

Je kan in Java ze beide implementeren met één datastructuur: ArrayDeque.

ArrayDeque = Array **D**ouble-**e**nded **que**ue

- Spreek uit als “Array deck”.
- Een queue die je aan beide kanten kan bereiken.

Functie	Head	Tail	Running time
Insert	`addFirst(e)`	`addLast(e)`	$O(1)$
Remove	`removeFirst(e)`	`removeLast(e)`	$O(1)$
Examine	`getFirst(e)`	`getLast(e)`	$O(1)$

Verder heb je nog een `contains(e)` functie die $O(n)$ runt.

4. ArrayDeque

FIFO en LIFO queues

Example code:

```
public static void main(String[] args) {
    ArrayDeque<String> rij = new ArrayDeque<>();
    rij.addLast("Joost");
    rij.addLast("Pieter");
    rij.addLast("Annabel");
    rij.removeFirst();

    System.out.print("Mensen in de wachtrij: ");
    for (String person : rij)
        System.out.print(person + " ");
}
```

Wat is de output?

```
public static void main(String[] args) {
    ArrayDeque<String> rij = new ArrayDeque<>();
    rij.addLast("Joost");
    rij.addLast("Pieter");
    rij.addLast("Annabel");
    rij.removeLast();

    System.out.print("Mensen in de wachtrij: ");
    for (String person : rij)
        System.out.print(person + " ");
}
```

Wat is de output?

4. ArrayDeque

FIFO en LIFO queues

Een ArrayDeque werkt achter de schermen met een array en twee integers:

1. Head → Index van de eerste van de queue
2. Tail → Index van de laatste van de queue

Als er een element aan de queue toegevoegd of verwijdert wordt, dan updaten deze indices. Bijvoorbeeld:

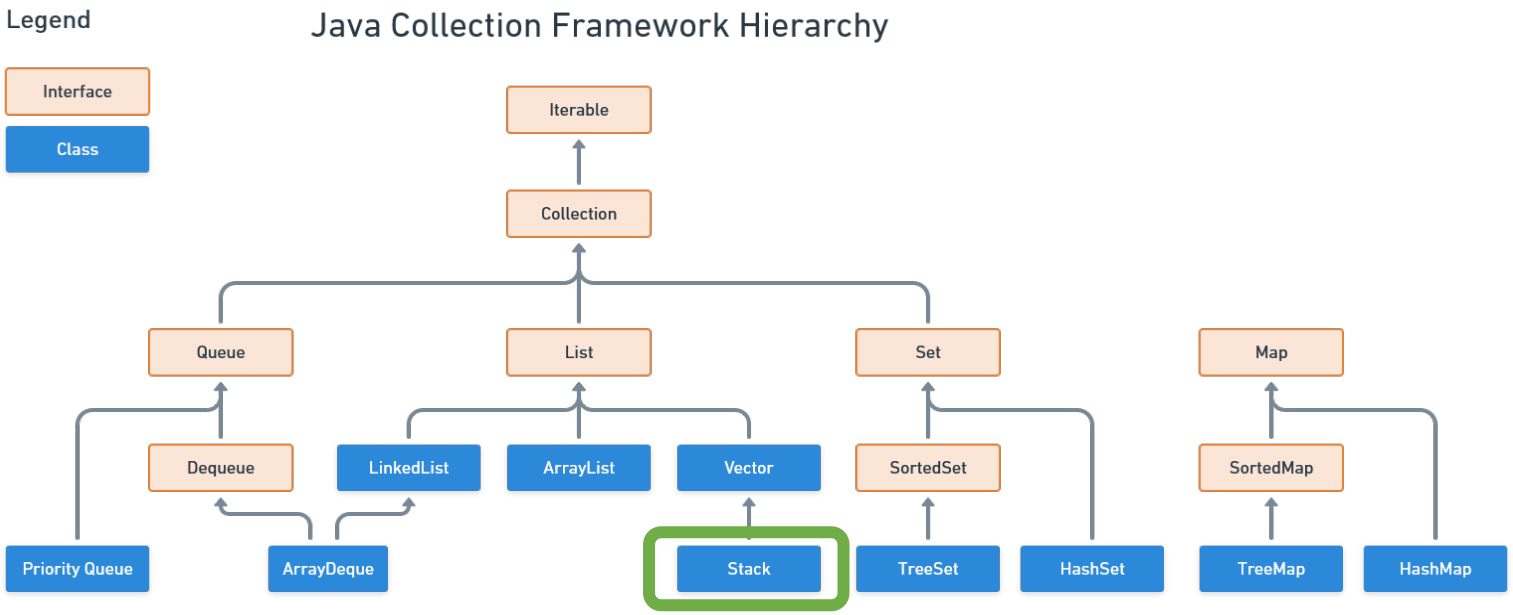
1. Als je de eerste element weghaalt, dan schuift de head index één naar rechts.
2. Als je een element achteraan toevoegt, dan schuift de tail index één naar rechts.

4. ArrayDeque

FIFO en LIFO queues

FAQ:

1. Wat is de Stack class binnen de Java Collection Framework?
   - Antwoord: Dat is een legacy class dat hetzelfde werkt als een LIFO queue, maar dan met andere benamingen: `push()`, `pop()` en `peek()`.



4. ArrayDeque

FIFO en LIFO queues

FAQ:

1. Wat is de Stack class binnen de Java Collection Framework?
   - Antwoord: Dat is een legacy class dat hetzelfde werkt als een LIFO queue, maar dan met andere benamingen: `push()`, `pop()` en `peek()`.
2. Kan je een LinkedList gebruiken in plaats van een ArrayDeque?
   - Antwoord: Ja!
     Het heeft dezelfde worst-case running times, maar in de praktijk is een ArrayDeque sneller en gebruikt minder memory.

4. ArrayDeque

Samenvatting

Queues kan je implementeren in Java met ArrayDeque.

FIFO = First in First out, zoals een normale rij.

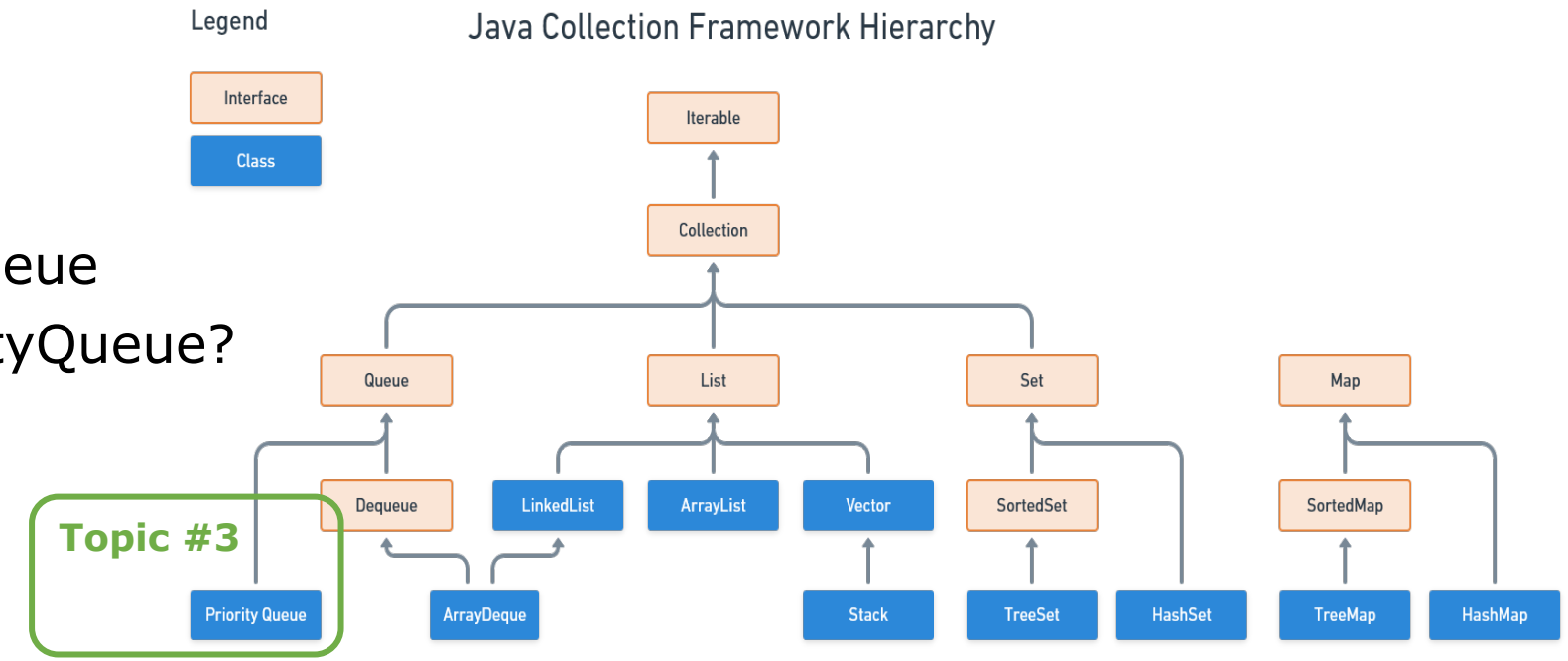
FILO = First in Last out, zoals een stack.

Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet	ArrayDeque
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**	$O(n)$
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**	$O(1)$
Toevoegen eerste	N/A	$O(n)$	$O(1)$	N/A	N/A	$O(1)$
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**	N/A
Verwijderen eerste	N/A	$O(n)$	$O(1)$	N/A	N/A	$O(1)$
Verwijderen laatste	N/A	$O(1)$	$O(1)$	N/A	N/A	$O(1)$

Les van vandaag

Inhoudsopgave

1. Opdrachten van vorige week
2. Set 2
3. Maps
4. ArrayDeque
5. **PriorityQueue**
   1. Wat is een PriorityQueue
   2. Hoe werkt een PriorityQueue?
6. Generics



5. Priority Queue

Wat is een Priority Queue

De laatste datastructuur van dit vak is de Priority Queue.

Met een Priority Queue kan je heel makkelijk de minimale of maximale waarde van je opgeslagen data vinden.

Je zou het ook kunnen zien als makkelijk de hoogste prioriteit eruithalen, vandaar de naam Priority Queue. Java implementeert het als een *min-heap*: de laagste value kan je peeken en pollen.

Functie	Head	Running time
Data toevoegen	`add(e)`	$O(\log n)$
Min value bekijken	`peek(e)`	$O(1)$
Min value verwijderen	`poll(e)`	$O(\log n)$

5. Priority Queue

Wat is een Priority Queue

Voorbeeld implementatie:

```
PriorityQueue<Vogelbekdier> prioQueue = new PriorityQueue<>();
for (Vogelbekdier vogelbekdier : vogelbekdieren) {
    prioQueue.add(vogelbekdier);
}
System.out.println("Eerste naam: " + prioQueue.peek().naam);
```

Resultaat: `Eerste naam: Elise`

5. Priority Queue

Wat is een Priority Queue

Hetzelfde als bij de TreeSet moet je de volgende dingen implementeren voor custom classes:

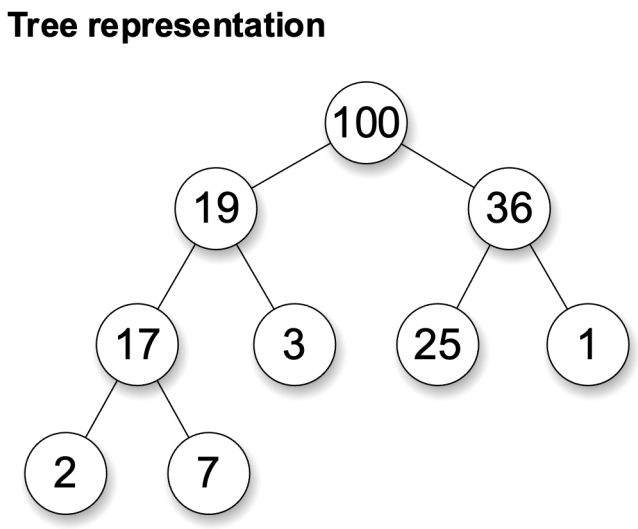
1. `equals()`
2. Natural ordering om te comparen
   1. `compareTo() [+ implements Comparable<>]`
   2. Een comparator meegeven bij het maken van de Priority Queue:

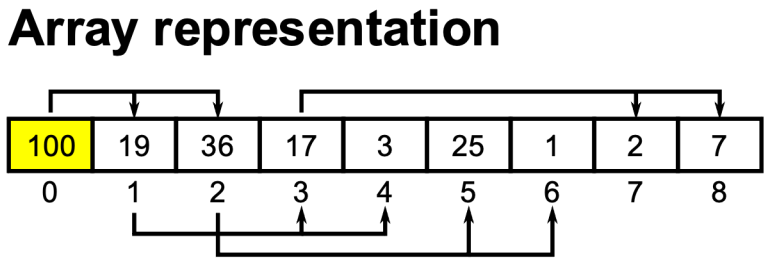
```
PriorityQueue<Vogelbekdier> prioQueue = new PriorityQueue<>( (e1, e2) ->
    Integer.compare(e1.lengte, e2.lengte)
);
```

5. Priority Queue

Wat is een Priority Queue

Achter de schermen wordt een Priority Queue geïmplementeerd als een **binary heap**.





Voor hoe de operaties werken:
https://en.wikipedia.org/wiki/Binary_heap

By Ermishin - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=12251273

5. Priority Queue

Wat is een Priority Queue

Priority Queue vs TreeSet

1. Priority Queue kan duplicates bevatten, TreeSet niet.
2. Running times:

Functie	PrioQueue (min)	PrioQueue (max)	TreeSet
Data toevoegen	$O(\log n)$	$O(\log n)$	$O(\log n)$
Min value bekijken	$O(1)$	N/A	$O(\log n)$
Min value verwijderen	$O(\log n)$	N/A	$O(\log n)$
Max value bekijken	N/A	$O(1)$	$O(\log n)$
Max value verwijderen	N/A	$O(\log n)$	$O(\log n)$

5. Priority Queue

Samenvatting

De Priority Queue is een datastructuur waarmee je makkelijk de laagste of

Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet	ArrayDeque	PrioQueue(min)
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**	$O(n)$	$O(n)$
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**	$O(1)$	$O(\log n)$
Toevoegen orde	N/A	$O(n)$	$O(1)$	N/A	N/A	$O(1)$	N/A
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**	N/A	N/A
Verwijderen orde	N/A	$O(n)$	$O(1)$	N/A	N/A	$O(1)$	N/A
Min bekijken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Min verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n)$
Max verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	N/A

5. Priority Queue

Opdrachten

1. Houd de maximale lengte van de Alpacas bij in een Priority Queue.
   1. Extract de lengte van de twee grootste alpacas.
   2. Poll de lengte van de grootste overgeleven alpacas.
2. Houd de volgorde van de Alpacas bij in een ArrayDeque + check functies
3. Stel je hebt een stuk code:

```
for (int i = 0; i < namen.length; i++) {
    vogelbekdieren[i] = new Vogelbekdier(lengtes[i], namen[i]);
}
```

Je wilt graag kijken of alle haakjes goed gesloten zijn in de goede volgorde. Voor de haakjes "(", "{", "[", bekijk of ze in de goede volgorde afgesloten worden.

Les van vandaag

Inhoudsopgave

1. Opdrachten van vorige week
2. Set 2
3. Maps
4. ArrayDeque
5. PriorityQueue

Les van vandaag

Samenvatting

Een Set is een verzameling van objecten, zonder duplicates die niet direct op index aangeroepen kunnen worden.

- HashSet → $O(1)$ → `.equals()` en `.hashCode()`
- TreeSet → $O(\log n)$, sorteert automatisch → `.equals()` en `.compareTo()`

Maps zijn datastructuren waarmee je makkelijk dingen kan opzoeken, in de vorm van Key → Value. Net als bij Sets heb je HashMaps en TreeMaps.

ArrayDeques zijn handig voor wanneer je orde van de data wilt bijhouden.

PriorityQueue is handig als je de *max* of *min* value continu wilt bijhouden.

Voor custom classes: `.equals()` en `.compareTo()`

Les van vandaag

Samenvatting

Operatie	Array	ArrayList	LinkedList	TreeSet	HashSet	ArrayDeque	PrioQueue(min)
Lezen/Updaten	$O(1)$	$O(1)$	$O(n)$	N/A	N/A	N/A	N/A
Zoeken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**	$O(n)$	$O(n)$
Toevoegen	N/A	$O(1)$*	$O(1)$	$O(\log n)$	$O(1)$**	$O(1)$	$O(\log n)$
Toevoegen orde	N/A	$O(n)$	$O(1)$	N/A	N/A	$O(1)$	N/A
Insertion	N/A	$O(n)$	$O(n)$	N/A	N/A	N/A	N/A
Verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$**	N/A	N/A
Verwijderen orde	N/A	$O(n)$	$O(1)$	N/A	N/A	$O(1)$	N/A
Min bekijken	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Min verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n)$
Max verwijderen	N/A	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	N/A

Les van vandaag

Opdracht

Bij de supermark Jumba ben jij verantwoordelijk om de kassa's en de rijen ervan te implementeren.

Er zijn 4 kassa's en elke kassa heeft een Rij aan mensen. Er komen continu mensen erbij en dan worden ze geholpen. Het duurt 10 seconde om iemand te helpen.

1. Implementeer een juiste datastructuur voor deze rij en zorg ervoor dat elke kassa met één rij geassocieerd wordt.

Daarnaast heeft Jumba een speciale actie: vier mensen in één rij? Een kassa erbij! Alle kassa's vol? Producten gratis!

2. Implementeer deze functionaliteit. Vul de kassa's van laag naar hoog om zo min mogelijk kassa's open te maken. Als alle kassa's vol zijn moet je printen dat de persoon zijn producten gratis krijgt.

Les van vandaag

Opdracht

Sample input

```
Jan 0
Katja 0
Piet 0
Hendrik 0
David 0
Sem 5
…
```

Sample output

```
Jan vertrekt om 10 bij kassa 1
Katja vertrekt om 10 bij kassa 1
Piet vertrekt om 10 bij kassa 1
Hendrik vertrekt om 10 bij kassa 2
David vertrekt om 10 bij kassa 2
Sem vertrekt om 15 bij kassa 2
...;
```