ALGORITMES & DATASTUCTUREN

Joost Visser (jwjh.visser@avans.nl)

Academie voor Deeltijd

Laatst geüpdatet: 4 mei 2021

Opdrachten en vragen

De antwoorden staan online:

1. Slides zijn geüpdatet
2. Code is online gezet
3. Sorting opdracht wordt doorgeschoven naar deze week

Voor deze week gaan we iets anders proberen: Patrick houdt de chat bij.

Les van vandaag

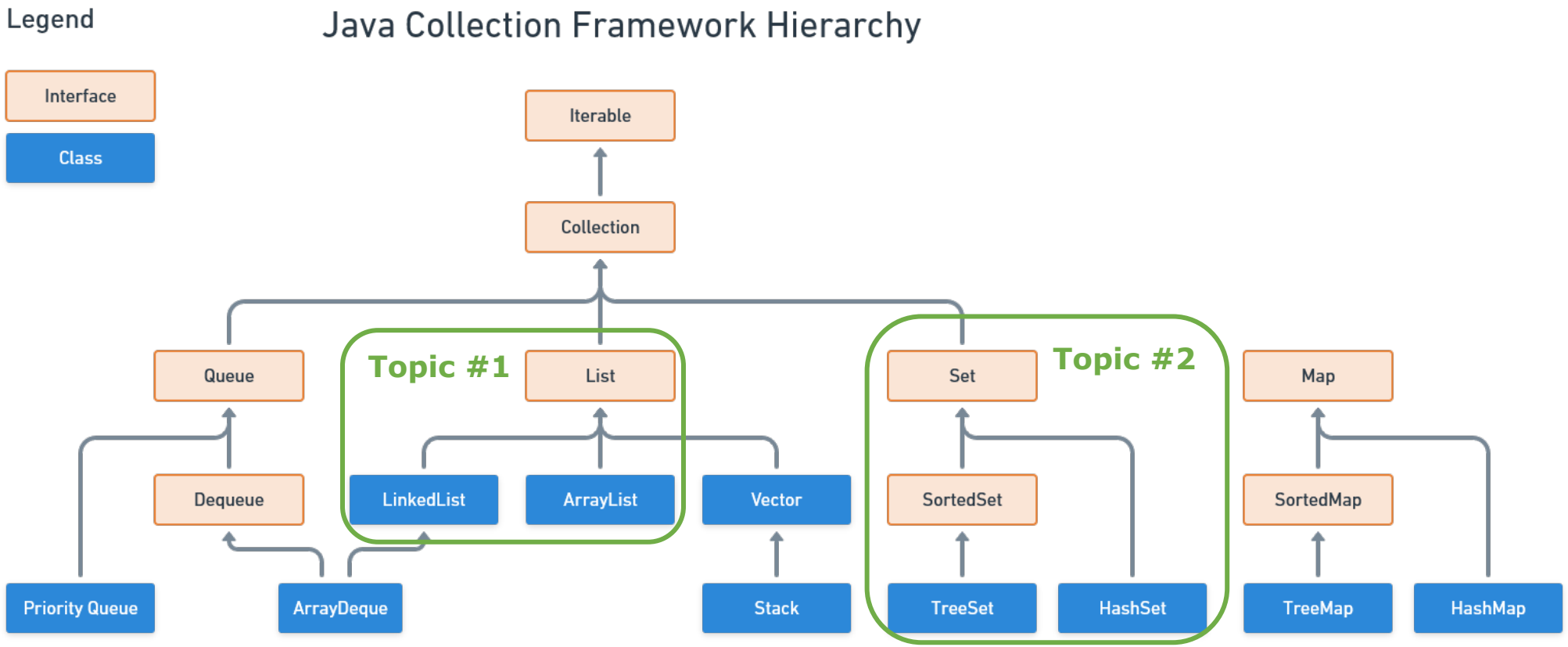
Lesplan

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, Map
3	18 mei 2021	Datastructuren 2	Stack, Heap, Deque, Priority Queue, Generieke datatypen
4	25 mei 2021	Algoritmes	Sorteren 2, recursie, binaire bomen, proeftoets

Vragen?

Les van vandaag

Java datastructuren



Les van vandaag

Inhoudsopgave

1. **Herhaling: Tijds- en geheugencomplexiteit**
    1. Wat te weten voor de toets
    2. Big-O intuitief uitgelegd
    3. Hoe bereken je het vanuit de code?
    4. Extra oefeningen
2. Voorbeeld: Sorteren
3. Arrays en Lists
4. Sets

1. Tijds- en geheugencomplexiteit

Wat te weten voor de toets

```java
/**
 * Finds a number in an unordered list.
 * Returns index or -1 if not found.
 */
public int findNumber(int list[], int number) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == number) {
            return i;
        }
    }
    return -1;
}
```

1. Gegeven een stuk simpele code:
   - *Wat is de worst-case verwerkingstijd?*
   - *Wat is het extra geheugengebruik?*
2. Je snapt wat de big-O notatie betekent
   - Als inputverzameling *n* 10x zo groot wordt, duurt het running van een $O(n^2)$ algoritme 100x zo lang.
3. Je kan verschillende snelheden vergelijken:
   - $O(n^2)$ is langzamer dan $O(n \log n)$ (voor grote n, in de worst-case).
   - $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n^2) < O(n^6) < O(2^n)$

1. Tijds- en geheugencomplexiteit

Big-O intuitief uitgelegd

Big-O is een notatie van groei wat we gebruiken om de snelheden van algoritmes te vergelijken. Dit noteren we met $T = O(n^2)$.

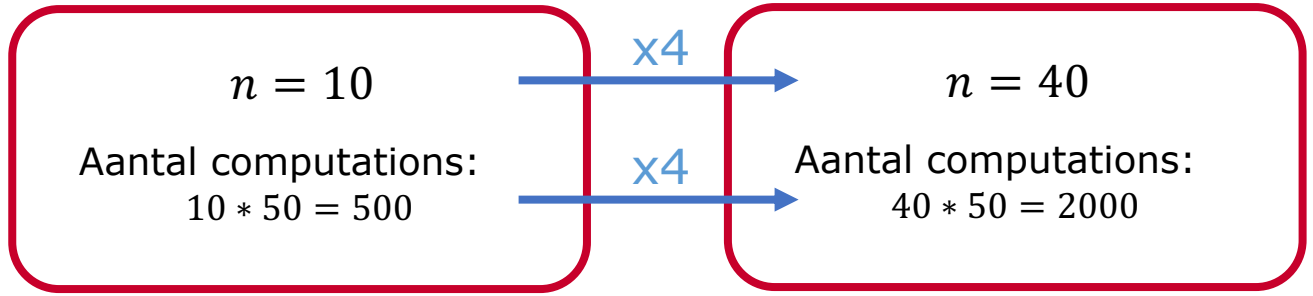
- $O(n^2)$: wordt de input p keer zo groot? → Aantal computations groeit met p^2 (worst-case)

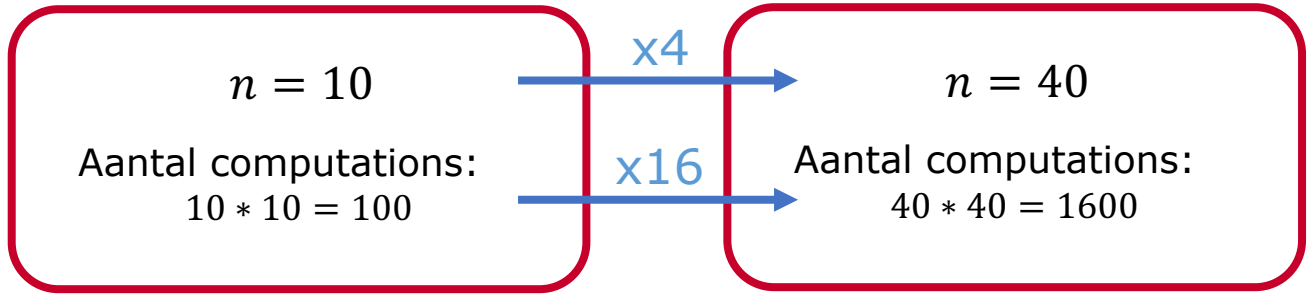
$T = O(n)$

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 50; j++) {
        step(i, j); // O(1)
    }
}
```

$T = O(n^2)$

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        step(i, j); // O(1)
    }
}
```





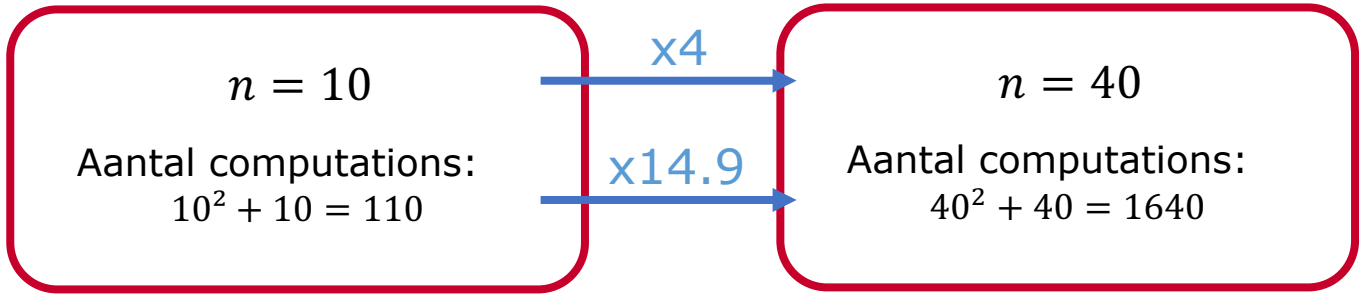
(Voor een wiskundigere uitleg, zie de slides van vorige week)

1. Tijds- en geheugencomplexiteit

Big-O intuitief uitgelegd

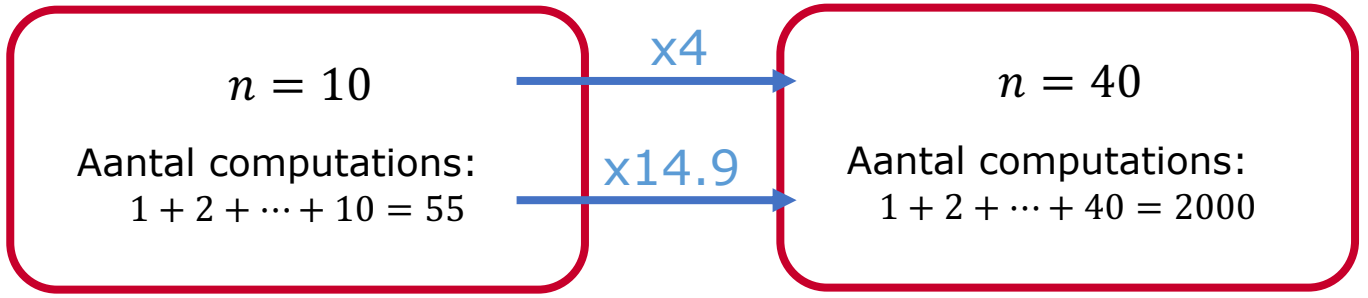
$T = O(n^2)$

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        step(i, j); // O(1)
    }
}
for (int i = 0; i < n; i++) {
    doSomething(i); // O(1)
}
```



$T = O(n^2)$

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        step(i, j); // O(1)
    }
}
```



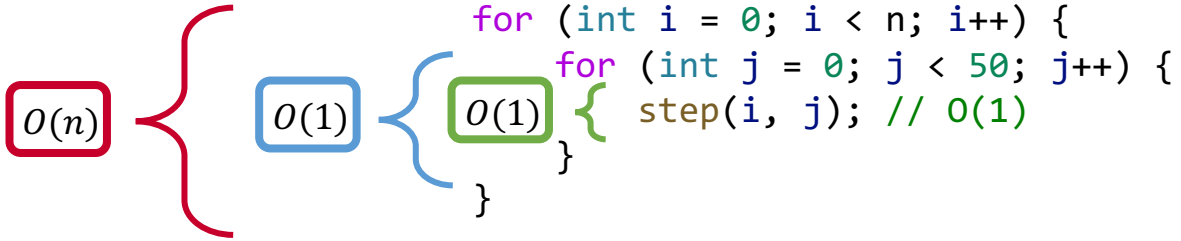
(Voor een wiskundigere uitleg, zie de slides van vorige week)

1. Tijds- en geheugencomplexiteit

Hoe bereken je het vanuit de code?

1. Handmatig kunnen tellen, zie vorige slides. (Niet exact, al is het voor nu goed genoeg)
2. Rekenen met Big-O notatie
   - $3 * O(n) = O(3n) \quad = O(n)$
   - $O(n^2) + O(n) = O(n^2 + n) \quad = O(n^2)$
   - $O(n) + O(\log n) = O(n + \log n) \quad = O(n)$
   - $O(n) * O(n) = O(n * n) \quad = O(n^2)$
   - $O(n) + O(n) = O(2n) \quad = O(n)$
   - $O(50) * O(1) = O(50) \quad = O(1)$

Voorbeeld:

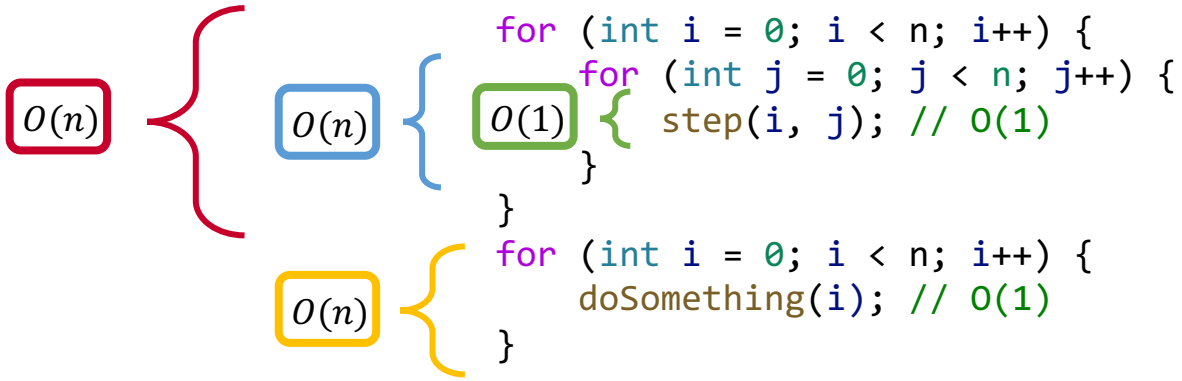


$$T = O(n) * O(1) * O(1) = O(n * 1 * 1) = O(n)$$

1. Tijds- en geheugencomplexiteit

Hoe bereken je het vanuit de code?

Nog een voorbeeld:



$$T = O(n) * O(n) * O(1) + O(n)$$
$$T = O(n * n * 1) + O(n)$$
$$T = O(n^2) + O(n)$$
$$T = O(n^2)$$

1. Tijds- en geheugencomplexiteit

Extra oefeningen

Oefening 1

```
for (int i = 0; i < 2n; i++) {
    doSomething(i); // O(1)
}
```

$O(n)$

Oefening 2

```
for (int i = n; i > 10; i--) {
    doSomething(i); // O(1)
}
```

$O(n)$

Oefening 3

```
for (int i = 0; i < n; i++) {
    if (i == 0) {
        for (int j = 0; j < n; j++)
            doSomething(i, j); // O(1)
    }
    else {
        doSomething(i); // O(1)
    }
}
```

$O(n)$

(Is equivalent aan oefening 3)

```
if (i == 0) {
    for (int j = 0; j < n; j++)
        doSomething(i, j); // O(1)
}

for (int i = 1; i < n; i++) {
    doSomething(i); // O(1)
}
```

1. Tijds- en geheugencomplexiteit

Wat te weten voor de toets

```java
/**
 * Finds a number in an unordered list.
 * Returns index or -1 if not found.
 */
public int findNumber(int list[], int number) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == number) {
            return i;
        }
    }
    return -1;
}
```

1. Gegeven een stuk simpele code:
   - *Wat is de worst-case verwerkingstijd?*
   - *Wat is het extra geheugengebruik?*
2. Je snapt wat de big-O notatie betekent
   - Als inputverzameling *n* 10x zo groot wordt, duurt het running van een $O(n^2)$ algoritme 100x zo lang.
3. Je kan verschillende snelheden vergelijken:
   - $O(n^2)$ is langzamer dan $O(n \log n)$ (voor grote n, in de worst-case).
   - $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n^2) < O(n^6) < O(2^n)$

Les van vandaag

Inhoudsopgave

1. Herhaling: Tijds- en geheugencomplexiteit
2. **Voorbeeld: Sorteren**
    1. Het sorteerprobleem
    2. Selection Sort
    3. Counting Sort
3. Arrays en Lists
4. Sets

2. Voorbeeld: Sorteren

Wat is het sorteerprobleem?

Sorteerprobleem: Herschik een gegeven array zodat de elementen in stijgende (of dalende) volgorde staan.

Voorbeelden:

1. Ongesorteerd: [19, 72, 44, 29, 44, 25, 18, 28, 93]
   Gesorteerd: [18, 19, 25, 28, 29, 44, 44, 72, 93]
2. Ongesorteerd: [“Piet”, “Jan”, “Katja”, “Annabel”, “Frans”, “Lieke”]
   Gesorteerd: [“Annabel”, “Frans”, “Jan”, “Katja”, “Lieke”, “Piet”]

Waarom analyseren we dit?

1. Makkelijk, duidelijk voorbeeld voor het illustreren van tijds- en geheugencomplexiteit.
2. Goed te zien wanneer een antwoord correct is.
3. Goed om algoritmes te vergelijken met elkaar.
4. In sommige situaties is het sneller om je eigen algoritmes te gebruiken.

2. Voorbeeld: Sorteren

Selection Sort

Input: [19, 72, 44, 29, 44, 25, 18, 28, 93]

Selection Sort v1

```
1. Create new outputArray[n]
2. For i = 0 to n-1:
    1. Find minValue in inputArray
    2. outputArray[i] = minValue
    3. Set minValue in inputArray
       to ∞
3. Return outputArray
```

Uitvoeringstijd? $O(n^2)$

```
1. O(n)
2. O(n)
    1. O(n)
    2. O(1)
    3. O(1)
3. O(1)
```

Extra geheugenverbruik?

$O(n)$

Kan dit beter?

Ja! Inline sorteren.
Extra geheugenverbruik van $O(1)$

Output: [18, 19, 25, 28, 29, 44, 44, 72, 93]

2. Voorbeeld: Sorteren

Selection Sort

Input: [19, 72, 44, 29, 44, 25, 18, 28, 93]

Selection Sort v2

```
1. For i = 0 to n-1:
    1. k = findMinValueIndex(i,n)
    2. Swap(i, k)
2. Return inputArray
```

Zie https://visualgo.net/en/sorting

Uitvoeringstijd? $O(n^2)$

```
1. O(n)
    1. O(n)
    2. O(1)
2. O(1)
```

Extra geheugenverbruik?

$O(1)$

Output: [18, 19, 25, 28, 29, 44, 44, 72, 93]

2. Voorbeeld: Sorteren

Selection Sort

Soortgelijke algoritmes

1. Insertion Sort
   - Sorteer van links naar rechts: eerste twee elementen, dan derde, dan vierde, enzovoorts.

2. Bubble Sort
   - Vergelijkbaar met Selection Sort, maar gebruikt meer swaps in plaats van zoeken naar de min. (Gaat ook uit van max → min sorteren)

Zie https://visualgo.net/en/sorting voor visualisaties.

2. Voorbeeld 2: Sorteren

Counting Sort

Input: [9, 2, 4, 9, 4, 5, 8, 8, 3]

Constraint: data zit tussen 0 en k. $(k = 9)$

Counting Sort

```
1. int[k] counts = new int[k]
2. For i = 0 to n-1:
    1. counts[inputArray[i]] ++;
3. i = 0;
4. for j = 0 to k-1:
    1. while (counts[j] > 0)
        1. inputArray[i] = j
        2. i ++;
        3. counts[j] --;
5. Return inputArray
```

Uitvoeringstijd?

1. O(1)
2. O(n)
3. O(1)
4. O(n+k)
5. O(1)

$O(n + k)$

Extra geheugengebruik?

$O(k)$

Nog vragen?

Output: [2, 3, 4, 4, 5, 8, 8, 9, 9]

2. Voorbeeld: Sorteren

Samenvatting

Sorteerprobleem: Herschik een gegeven array zodat de elementen in stijgende (of dalende) volgorde staan.

Selection sort: Van links naar rechts, vind de minimale element en zet het links neer.

- Uitvoeringstijd: $\mathrm{T(n)} = O(n^2)$
- Extra geheugengebruik: $G(n) = O(1)$

Counting sort: Tel van 0 tot k hoe vaak ze voorkomen.

- Uitvoeringstijd: $\mathrm{T(n)} = O(n + k)$
- Extra geheugengebruik: $G(n) = O(k)$
- Werkt alleen als alle getallen tussen de 0 en k zitten.

Bekijk https://visualgo.net/en/sorting voor visualisaties van de algoritmes.

Opdracht

Opdracht: Sorteren van vogelbekdieren

Perry vindt het leuk om vogelbekdieren te houden. Er komt een nieuwe lading vogelbekdieren binnen en hij zou ze graag willen sorteren op naam en op lengte.

Implementeer deze functionaliteit met je eigen sorteerfunctie!

Input: array aan Vogelbekdieren [vogelbekdier1, vogelbekdier2, …]

- String vogelbekdier1.naam bevat de naam van het vogelbekdier1
- int vogelbekdier1.lengte bevat de lengte in cm van het vogelbekdier1.

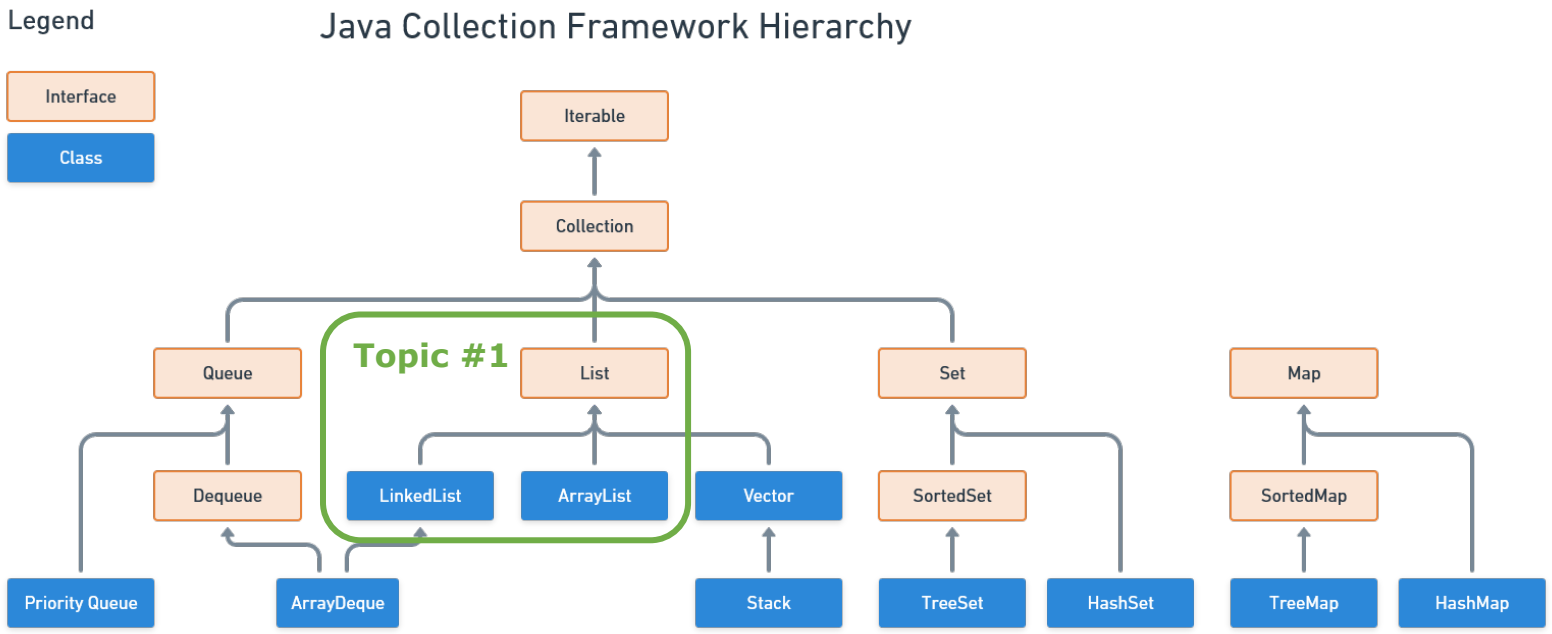
Output (print naar System.out):

- Array van vogelbekdieren, gesorteerd op naam
- Array van vogelbekdieren, gesorteerd op lengte

Les van vandaag

Inhoudsopgave

1. Herhaling: Tijds- en geheugencomplexiteit
2. Voorbeeld: Sorteren
3. Arrays en Lists
   1. Arrays
   2. ArrayList
   3. LinkedList
   4. Abstract Data Type
4. Sets



3. Arrays en Lists

Arrays

We hebben deze les al meerdere malen arrays gebruikt.

Een array is een structuur voor opslag van een lijst van elementen:

1. Dezelfde type
2. Aaneengesloten aan het geheugen
3. Lengte van de lijst vooraf bepaald

```
int[] intArray = new int[10];
```

Hoe zit dit eruit?

3. Arrays en Lists

Arrays

Hoe zit dit eruit?

```
int[] intArray = new int[10];
```

Index	Waarde
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

3. Arrays en Lists

Arrays

Hoe zit dit eruit?

```
int[] intArray = new int[10];
intArray[2] = 28;
```

Index	Waarde
0	0
1	0
2	28
3	0
4	0
5	0
6	0
7	0
8	0
9	0

3. Arrays en Lists

Arrays

Wat zijn de functionaliteiten van een array?

Operatie		Array
Lezen/Updaten	Op index	$O(1)$
Zoeken	Linear	$O(n)$
Zoeken	Binair (lijst gesorteerd)	$O(\log n)$
Toevoegen	Nieuwe elementen	N/A
Verwijderen		N/A

Nadeel: array is fixed lengte, dus extra elementen inserten gaat niet. Dit is onhandig, kan dit beter?

Index	Waarde
0	0
1	0
2	28
3	0
4	0
5	0
6	0
7	0
8	0
9	0

3. Arrays en Lists

ArrayList

Dat kan met een **ArrayList**!

Een ArrayList is een klasse die automatisch de arrays voor je beheert.

Je kan het beschouwen als een array met variabele lengte.

Achter de schermen:

1. Houdt het een array bij met fixed lengte.
2. Als je iets toe wilt voegen en de array zit vol, dan kopieert het naar een nieuwe array.

3. Arrays en Lists

ArrayList

Dat kan met een **ArrayList**!

Een ArrayList is een klasse die automatisch de arrays voor je beheert.

Je kan het beschouwen als een array met variabele lengte.

Achter de schermen:

1. Houdt het een array bij met fixed lengte.
2. Als je iets toe wilt voegen en de array zit vol, dan kopieert het naar een nieuwe array.

3. Arrays en Lists

ArrayList

Functies van een **ArrayList:**

Operatie		ArrayList
`get(i)`		
`set(i)`		
`add(e)`	Bij niet-volle lijst	
`add(e)`	Bij volle lijst	
`add(i,e)`	Altijd dooschuiven	
`remove(i)`	En doorschuiven	
`contains(e)`		

*Als de orde niet uitmaakt, zou je een eigen implementatie kunnen maken van *remove(i)* van $O(1)$.

3. Arrays en Lists

ArrayList

Hoe werk je met een ArrayList?

Omdat het een klasse is, moet je het ook aanroepen met functies in plaats van met de ingebakken array notatie.

Array

```
int[] intList = new int[10];
intList[3] = 420;
int a = intList[3];
```

ArrayList

```
ArrayList<Integer> intList = new ArrayList<>();
intList.set(3, 420);
int a = intList.get(3);
```

3. Arrays en Lists

ArrayList

Laten we het tabelletje aanvullen.

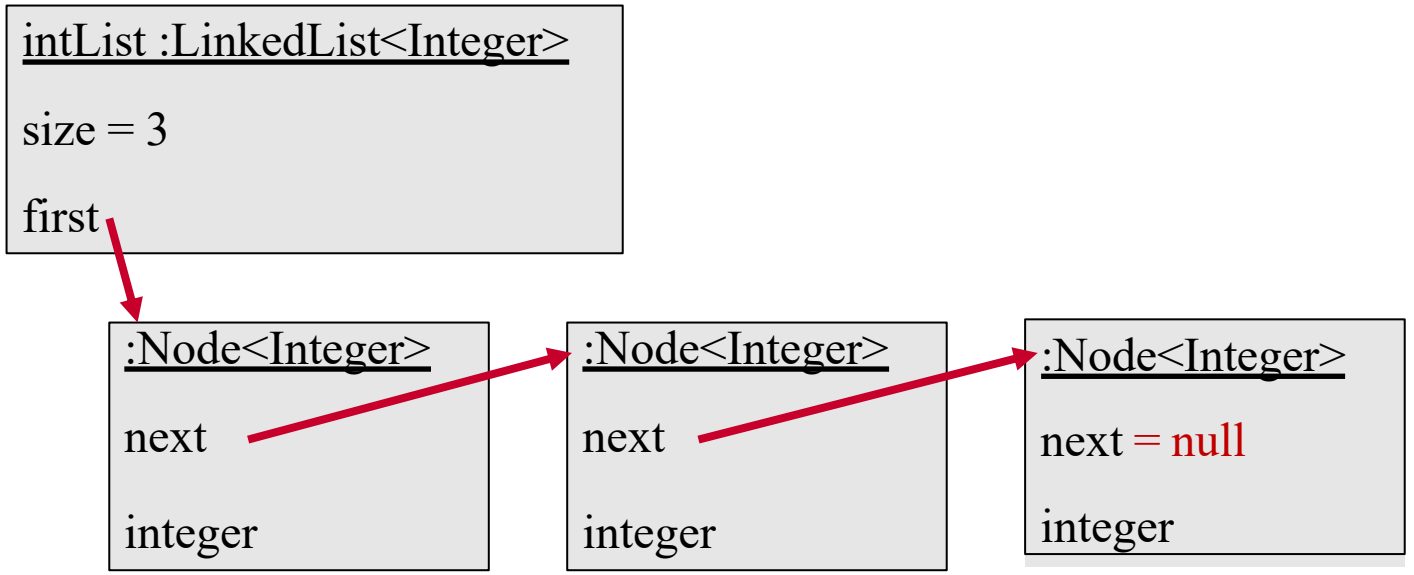
Operatie		Array	ArrayList
Lezen/Updaten	Op index	$O(1)$	
Zoeken	Linear	$O(n)$	
Zoeken	Binair (lijst gesorteerd)	$O(\log n)$	
Toevoegen	Nieuwe elementen	N/A	
Toevoegen	Bij volle lijst	N/A	
Verwijderen	En aanshuiven	N/A	

3. Arrays en Lists

LinkedList

Het kan ook op een andere manier, met de welbekende **LinkedList**.

In plaats van alle getallen achter elkaar te zetten, bewaar je de referentie (locatie) naar het volgende object.



3. Arrays en Lists

LinkedList

Waarom wil je dit?

1. Inserten van getallen tussen plekken kan een stuk sneller.
2. Deleten gaat ook een stuk sneller.
3. Heeft, net als een ArrayList, ook een dynamische lengte.

Hoe werk je ermee?

- Hetzelfde als een ArrayList, aangezien het beide de List interface implementeert!

ArrayList

```
ArrayList<Integer> intList = new ArrayList<>();
intList.set(3, 420);
int a = intList.get(3);
```

LinkedList

```
LinkedList<Integer> intList = new LinkedList<>();
intList.set(3, 420);
int a = intList.get(3);
```

3. Arrays en Lists

LinkedList

Functies van een **LinkedList:**

Operatie		LinkedList
get(i)		
set(i)		
add(e)	Bij niet-volle lijst	
add(i,e)	Zoeken niet meegerekend	
add(i,e)	Zoeken wel meegerekend	
remove(i)	Zoeken niet meegerekend	
remove(i)	Zoeken wel meegerekend	
contains(e)		

*Als je een element moet toevoegen of verwijderen moet je het in de praktijk altijd zoeken, waardoor je op $O(n)$ uitkomt. Echter, mocht je met .iterator() werken, dan kan het daadwerkelijk $O(1)$ zijn.

3. Arrays en Lists

LinkedList

Laten we het tabelletje aanvullen.

Operatie		Array	ArrayList	LinkedList
Lezen/Updaten	Op index	$O(1)$	$O(1)$	
Zoeken	Linear	$O(n)$	$O(n)$	
Zoeken	Binair (lijst gesorteerd)	$O(\log n)$	$O(\log n)$	
Toevoegen	Nieuwe elementen	N/A	$O(1)$	
Toevoegen	Bij volle lijst	N/A	$O(n)$	
Insertion	Op index	N/A	$O(n)$	
Verwijderen	En aanshuiven	N/A	$O(n)$	

3. Arrays en Lists

ArrayLists

+ Get(i) en set(i) zijn $O(1)$
+ add(i) is gemiddeld sneller
- add(i) is af en toe $O(n)$
+ Kan sorteren en binair zoeken
- Eerste element insert/delete is $O(n)$
- Insert/delete in het midden is $O(n)$

LinkedLists

- Get(i) en set(i) zijn $O(n)$
- add(i) is gemiddeld langzamer
+ add(i) is altijd O(1)
- Zoeken is altijd $O(n)$, zelfs gesorteerd
+ Kan snel eerste element inserten/deleten
+ Kan tijdens .iterate() in $O(1)$ insert/delete

Wanneer gebruik je wat?

In de praktijk is bijna altijd een **ArrayList** sneller:

1. get() en set() zijn veel sneller, add() is een beetje sneller
2. Eerste element inserten/deleten kan beter met een ArrayDeque (volgende lecture)

Alleen voor tijdens .iterate() kan het gebruik van LinkedList voordeel leveren.

3. Arrays en Lists

Abstract Data Types (ADTs)

Als we naar ArrayList en LinkedList kijken, wat valt op?

ArrayList

```
ArrayList<Integer> intList = new ArrayList<>();
intList.set(3, 420);
int a = intList.get(3);
```

LinkedList

```
LinkedList<Integer> intList = new LinkedList<>();
intList.set(3, 420);
int a = intList.get(3);
```

Ze implementeren precies dezelfde interface, hebben dezelfde functionaliteit maar zitten onder de schermen erg anders in elkaar.

Ze zijn beide Abstracte Data Typen → Zet hoe je de data opslaat los van de functionaliteit, zolang het maar een aantal publieke functies implementeert is het goed.

3. Arrays en Lists

Abstract Data Types (ADTs)

Een schaakbord zou je bijvoorbeeld op kunnen slaan als een 2D 8x8 array met Pieces op elk plek.

Of het zou kunnen met één 64 int[] array met schaakstukken.

(Dit kan zelfs nog efficienter.)

Zolang ze maar de functies movePiece(), resetBoard() etc implementeren, maakt het niet uit hoe ze de data opslaan.

Opdrachten

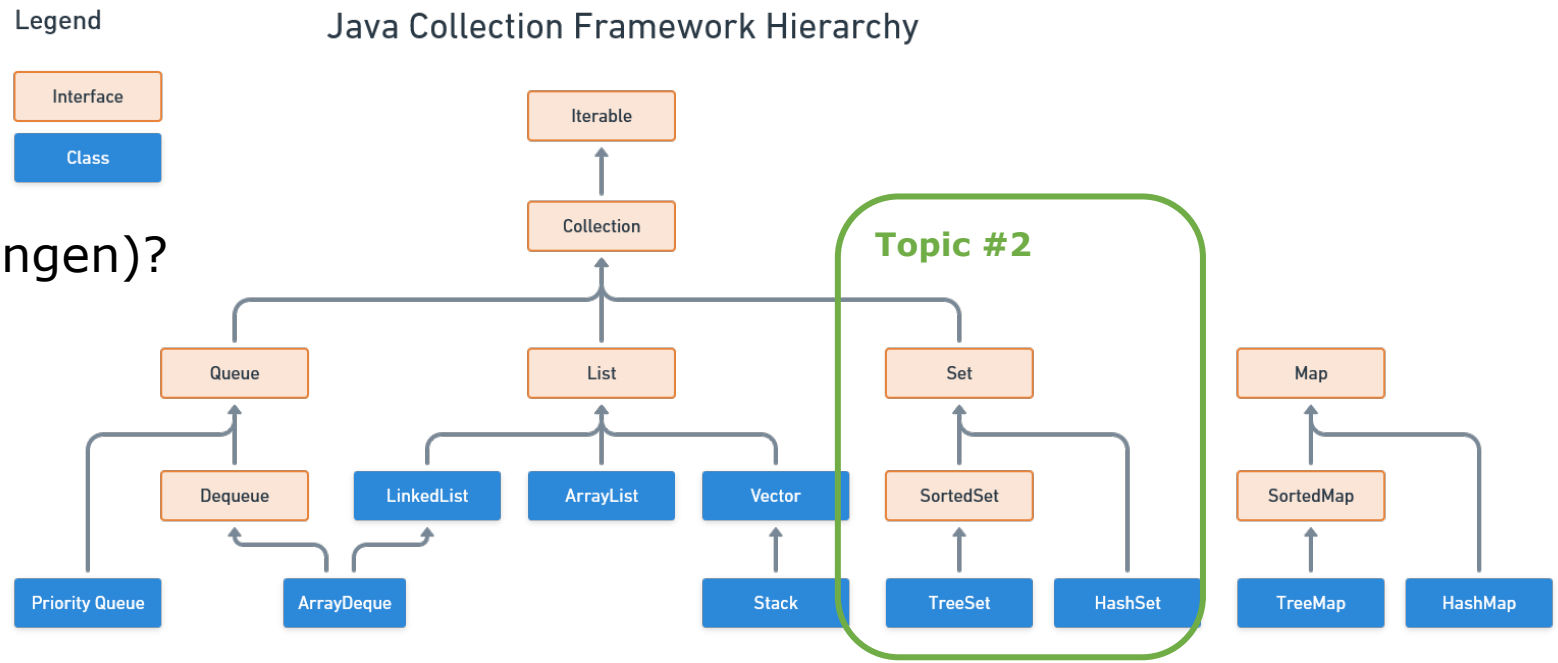
ArrayList en LinkedList oefenopdrachten

1. Implementeer “Sorteren van vogelbekdieren” opdracht met ArrayLists.
   Je mag gebruik maken van sorteerfuncties in Java.
2. [Interview vraag] Implementeer je eigen LinkedList.
3. [Optioneel] Implementeer je eigen ArrayList.

Les van vandaag

Inhoudsopgave

1. Herhaling: Tijds- en geheugencomplexiteit
2. Voorbeeld: Sorteren
3. Arrays en Lists
4. Sets
    1. Wat zijn sets (verzamelingen)?
    2. TreeSets
    3. HashSets



4. Sets

Wat zijn sets (verzamelingen)?

We hebben een ArrayList en LinkedList gezien, daarmee is zo'n beetje alles mogelijk om op te slaan wat we willen.

Waarom andere datastructuren?

1. Sommige datastructuren zijn gespecialiseerd in sommige functies (zoals Queues, die goed zijn in de volgorde van data bij te houden)
2. Soms kan je, als je sommige aannames kan maken, een stuk snellere datastructuur gebruiken.

#2 → Sets

4. Sets

Wat zijn sets (verzamelingen)?

Aanname van sets:

1. Orde van data maakt niks uit
2. Gedupliceerde data hoeft niet bijgehouden te worden. (We hoeven niet bij te houden of Jan 1x of 10x is geweest.)

Wanneer is dit het geval?

1. Bij het bijhouden welke steden je al bezocht hebt.
2. Presentielijsten van klasses
3. Databases (Key → Value)

4. Sets

Wat zijn sets (verzamelingen)?

De volgende functies van sets zijn belangrijk:

Operatie		ArrayList	LinkedList
contains(e)	Zit e in de zet?	$O(n)$	$O(n)$
add(e)	Voeg e aan de set toe.	$O(1)$	$O(1)$
remove(e)	Haal e uit de set weg.	$O(n)$	$O(n)$

Kan dit sneller? → Ja!

4. Sets

Wat zijn sets (verzamelingen)?

Er zijn over het algemeen twee sets:

1. HashSet
2. TreeSet

We beginnen met TreeSet.

Operatie	
contains(e)	Zit e in de zet?
add(e)	Voeg e aan de set toe.
remove(e)	Haal e uit de set weg.

4. Sets

TreeSet

TreeSet implementeert een *self-balancing binary search tree.* (Red-black tree)

→ We leggen in les 4 uit hoe een binary search tree werkt.

Extra voordeel: objecten zijn altijd gesorteerd in een TreeSet.

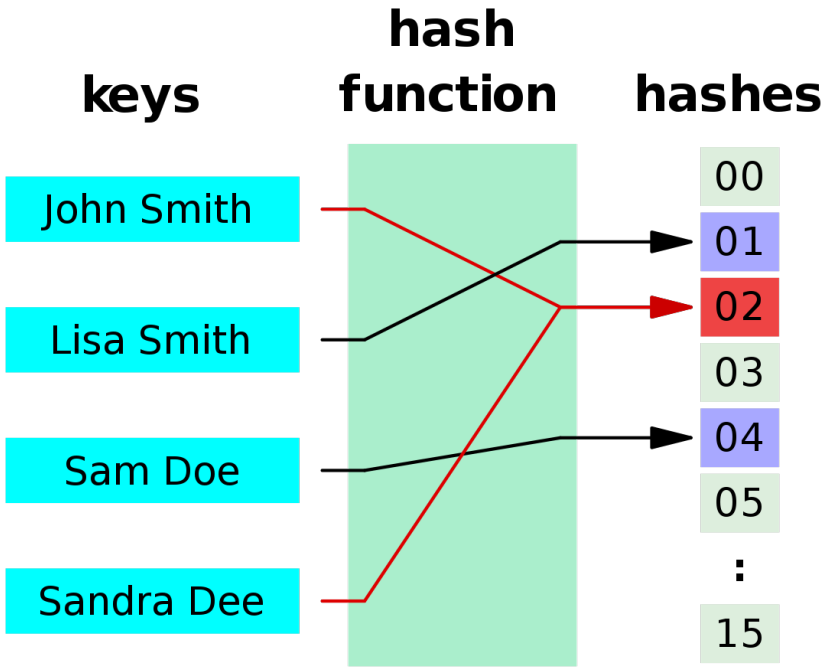
Operatie		TreeSet
contains(e)	Zit e in de zet?	$O(\log n)$
add(e)	Voeg e aan de set toe.	$O(\log n)$
remove(e)	Haal e uit de set weg.	$O(\log n)$

4. Sets

HashSet

Een HashSet gebruikt hashing:

1. Map alle objecten naar een uniek getal, de *hashCode* (32 bits int).

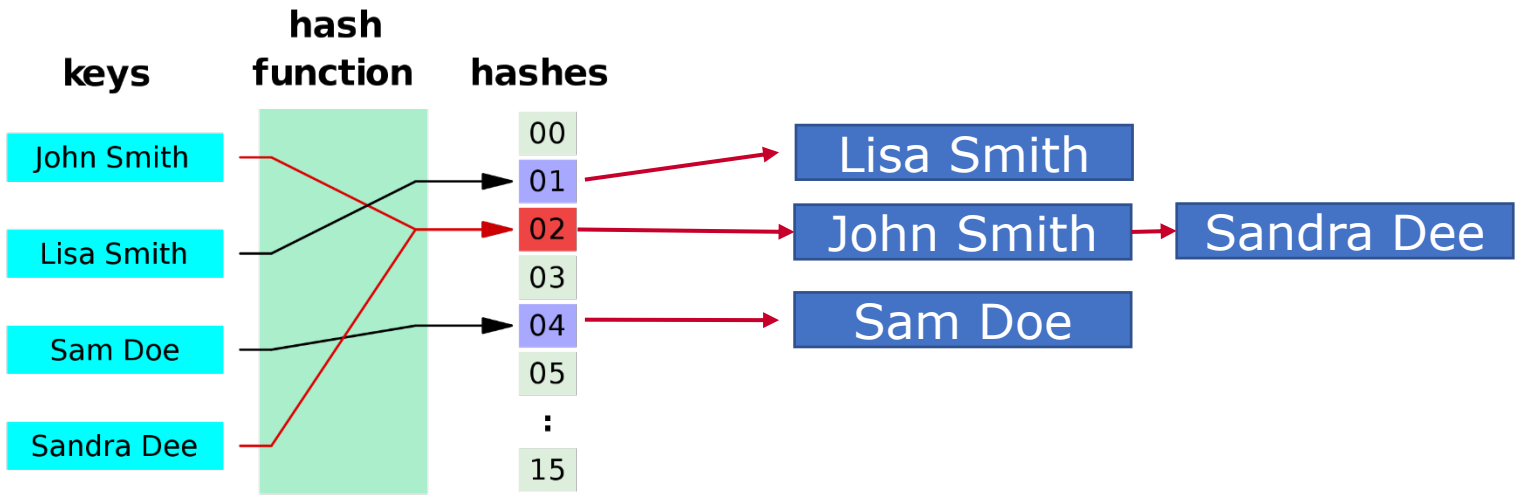


4. Sets

HashSet

Een HashSet gebruikt hashing:

1. Map alle objecten naar een uniek getal, de *hashCode* (32 bits int).
2. Functie add(e):
   1. Bereken de hashCode van object e.
   2. Voeg object toe aan lijst (LinkedList) bij deze hashcode.



4. Sets

HashSet

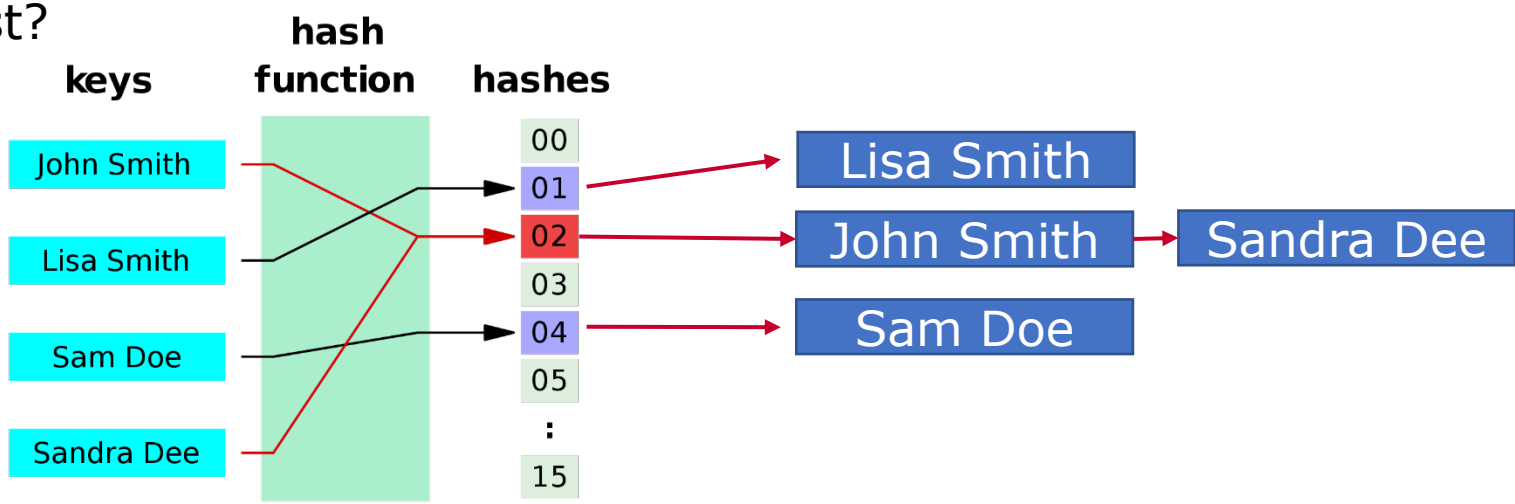
Functie contains(e):

1. Bereken de hashCode van object e.
2. Is de hashcode nog eerder voorgekomen? → return false
3. Is de hashcode wel eerder voorgekomen? → Check voor *collisions*
   1. Zit object in LinkedList (.equals())? → return true
   2. Zit object niet in LinkedList?

Joost Visser .hashCode() = 5

Sandra Dee .hashCode() = 2

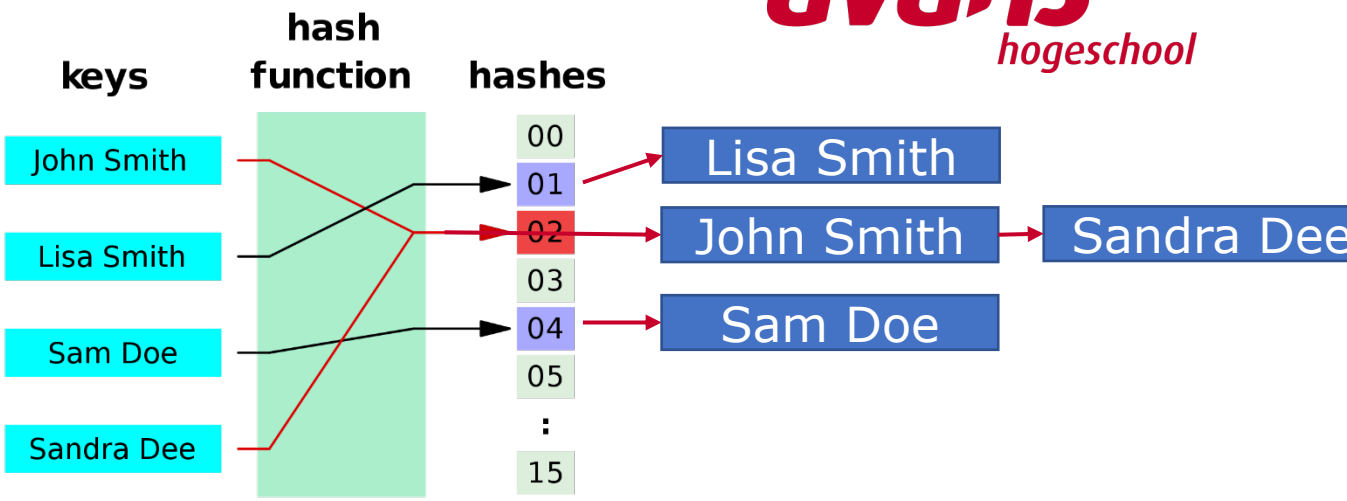
Eva Molenaar .hashCode() = 2



4. Sets

HashSet

Hoe snel is een HashSet?



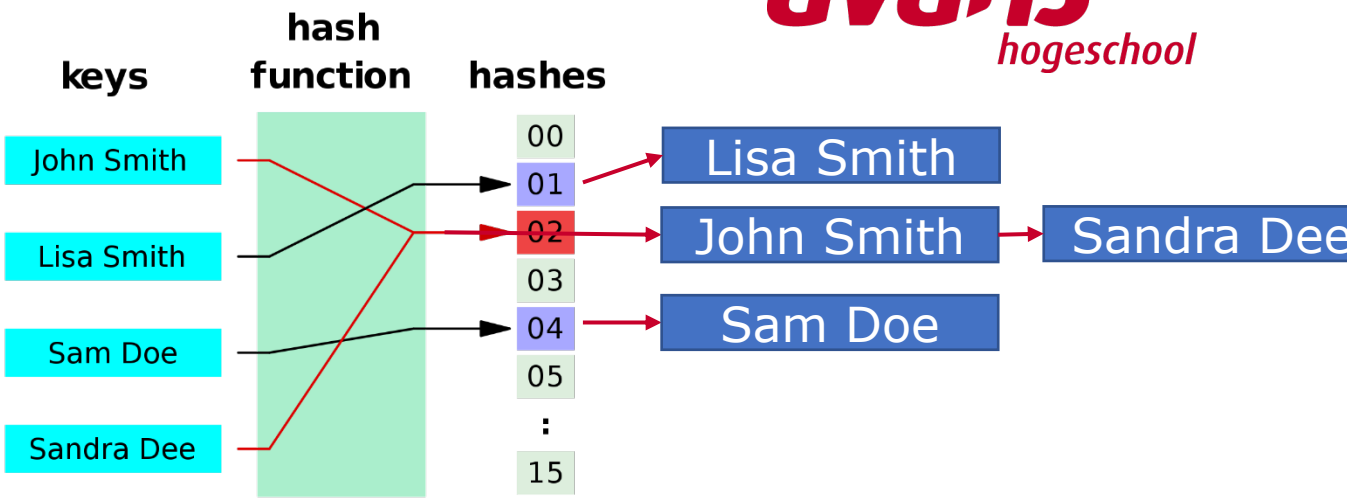
Operatie		TreeSet	HashSet
contains(e)	Zit e in de zet?	$O(\log n)$	$O(1)$*
add(e)	Voeg e aan de set toe.	$O(\log n)$	$O(1)$*
remove(e)	Haal e uit de set weg.	$O(\log n)$	$O(1)$*

*Hangt af van het aantal collisions. Als alles collide op één hash dan is het $O(n)$ worst-case.

4. Sets

HashSet

Hoe snel is een HashSet?



Operatie		TreeSet	HashSet
contains(e)	Zit e in de zet?	$O(\log n)$	$O(1)$*
add(e)	Voeg e aan de set toe.	$O(\log n)$	$O(1)$*
remove(e)	Haal e uit de set weg.	$O(\log n)$	$O(1)$*

*Hangt af van het aantal collisions. Als alles collide op één hash dan is het $O(n)$ worst-case.

4. Sets

TreeSets en HashSets met custom classes

Als je HashSets en TreeSets voor bestaande datatypes wilt gebruiken als integer en Strings, dan kan dit gewoon prima.

Echter, als je je eigen classes wilt gebruiken, dan moet je wat extras implementeren:

1. Voor HashSet: .equals en .hashCode()
2. Voor TreeSet: .equals en .compareTo()
   1. Voor het tweede heb je: “class A implements Comparable<A>” nodig.

```
public int hashCode()
public boolean equals(Object obj)
public int compareTo(Vogelbekdier o)
```

Return unieke hashCode getal.

Return true als objecten gelijk zijn.

Return negatief als object kleiner is dan o, 0 als ze gelijk zijn en positief als huidige object groter dan object o is.

4. Sets

Opdrachten

1. Nog een sorteermanier: Voeg alle vogelbekdieren aan een TreeSet toe en iterate dan over de TreeSet heen.
2. Voeg alle vogelbekdieren toe aan een HashSet en check of alle functies .contains(), .add() en .remove() werken.