Algorithmen und Datenstrukturen
Kapitel 3: Elementare Datenstrukturen

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Motivation

Quelle: [5]

- Wie findet man ***schnell*** Daten?
- Wie organisiert man Daten, damit ein Algorithmus schnell arbeiten kann?

Inhalt

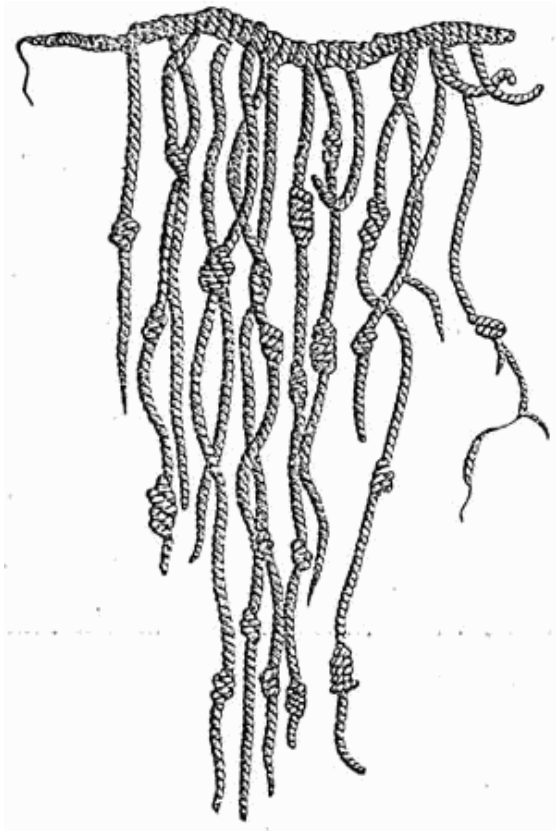
- **Datenstrukturen**
- Listen
- Stacks, Queues
- Mengen

Brainstorming: Datenstrukturen?

- Arrays, auch mehrdimensional
- Verkettete Listen
- Maps, assoziative Arrays, Dictionary
- Bäume
- Heaps
- Matrizen
- …

Datenstrukturen

- **Definition: Datenstruktur**
  - *Eine **bestimmte Art, Daten zu verwalten** und miteinander zu verknüpfen, um in geeigneter Weise auf diese zugreifen und diese manipulieren zu können. Datenstrukturen sind immer mit bestimmten **Operationen** verknüpft, um eben diesen Zugriff und diese Manipulation zu ermöglichen.*



Quipu: Eine der ersten Datenstrukturen [3]

- Wahl der Datenstruktur beeinflusst **Laufzeit des Algorithmus**
  - Datenstruktur muss für die konkrete Aufgabe bzw. für typische Eingaben optimiert sein.

Abstrakte Datentypen, Datenstrukturen

- ***Abstrakter Datentyp (ADT)***
  - Verbund an Daten zusammen mit der Definition aller zulässigen Operationen, die auf sie zugreifen. (*Wikipedia*)
  - API bzw. Schnittstelle beschreibt ***WAS*** die Operationen tun, aber nicht ***WIE***.

- ***Datenstruktur***
  - Realisierung bzw. Implementierung eines Abstrakten Datentyps (ADT)
  - Erst hier wird das ***WIE*** spezifiziert.

- Ein ADT kann durch verschiedene Datenstrukturen implementiert werden.

Überblick: Wichtige ADTs

- ***Liste / Sequenz (engl. "List")***
  - Geordnete Werte, Werte können doppelt vorkommen.
  - Beispiel: 3-6-2-5-3-1
  - Typische Operationen: `add, remove, size, isEmpty`
- ***Menge (engl. "Set")***
  - Reihenfolge ohne Bedeutung, meist darf jeder Wert nur einmal vorkommen.
  - Beispiel: {2, 3, 4, 5}
  - Typische Operationen `add, remove, size, isEmpty, contains`
- ***Map, Dictionary (dt. "assoziatives Datenfeld")***
  - Speichert Key-Value Pairs, Schlüssel-Werte Paare
  - Beispiel: Alter von Personen → { (Trump, 73), (Merkel, 65), (Kurz, 33) }
  - Typsiche Operationen: `insert, remove, size, isEmpty, containsKey, containsValue`
- ***Stack (dt. "Stapel" / "Keller")***
  - Spezielle Liste, Zugriff nur auf den zuletzt hinzugefügten Wert.
  - Typische Operationen: `push, pop, peek, size`
- ***Queue (dt. "Warteschlange")***
  - Spezielle Liste, Zugriff nur auf den ältesten Wert.
  - Typische Operationen: `enqueue, dequeue, tail, head, size`

Überblick: Wichtige Datenstrukturen

- ***Arrays***
  - Geeignet zur Implementierung der ADTs Listen, Stacks und Queues
- ***Verkettete Listen***
  - Geeignet zur Implementierung der ADTs Listen Stacks und Queues

Jetzt!

- ***Hashtabellen***
  - Geeignet zur Implementierung der ADTs Set und Maps
- ***Bäume***
  - Geeignet zur Implementierung der ADTs Set und Maps

Spätere Kapitel

- ***Graphen***
  - Abstraktion von Beziehungen
- ***Anwendungsspezifische Datenstrukturen***
  - Elektrische Schaltungen, Genom, Multimedia

Wiederverwendbarkeit

- Nutzung existierender Implementierung in Form von Klassen, Klassenbibliotheken, Packages
- Ggfs. Erweiterung durch Vererbung und Einbettung.
- Beispiele
  - Java Collection Framework
  - JGraphT: Graph Library, http://jgrapht.org/javadoc/
  - Boost C++ Bibliotheken: https://www.boost.org
  - TensorFlow, z.B. für maschinelles Lernen: https://www.tensorflow.org/
- **Aber auch: Eigenständige Implementierung**
  - Insbesondere für anwendungsspezifische Datenstrukturen
  - Ausnutzung der Anwendungssemantik führt meist zu effizienteren Lösungen.

Inhalt

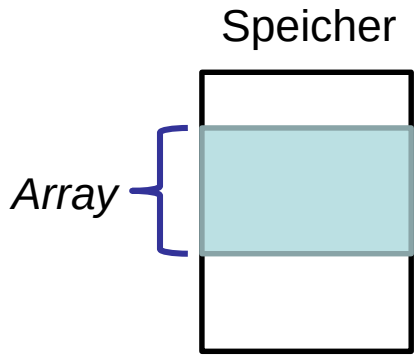
- Datenstrukturen
- **Listen**
- Stacks, Queues
- Mengen

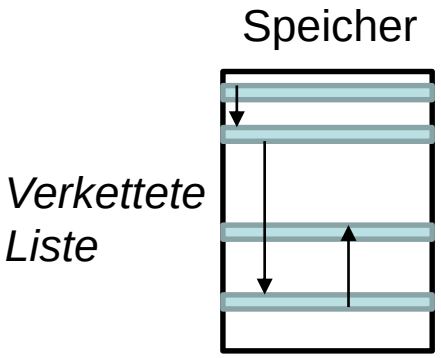
ADT: Liste / Sequenz

- **ADT Liste**
  - *Endliche, **geordnete*** Folge von Elementen eines *Grundtyps.*
  - **Elemente *x*** können öfters vorkommen.

- **Grundoperationen**
  - `add(x)`: Fügt x am Ende der Liste hinzu.
  - `remove(i)`: Löscht Element mit Index `i` aus der Liste.
  - `indexOf(x)`: Suche, liefert "Index" des ersten Vorkommens zurück und -1 falls nicht in Liste.
  - `size`: Anzahl der Listenelemente
  - `isEmpty()`: Gibt `true` zurück, falls Liste leer

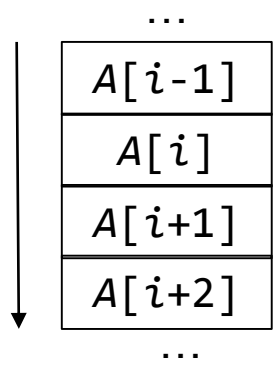
- **2 Möglichkeiten der Umsetzung**
  - ***Array***: Elemente in zusammenhängendem Speicherbereich abgelegt; Zugriff auf das *i*.-te Element über Adressrechnung
  - ***Verkettete Liste*****:** Listenelemente sind verstreut im Speicher; Zusammenhang über “Zeiger”





ADT Liste als Array

- **Array *A***
  - Daten liegen ***sequentiell*** im Speicher.
  - Falls jedes Datum 1 Byte → Daten liegen bei *N*, *N*+1, … *N*+(*n*-1)



- **Zugriff** auf *i*.tes Element über ***Adressrechnung***
  - Element A[i] steht an Adresse *N*+(*i-1*)
  - Achtung: Das erste Element steht an Index 0!

- **Implementierung der Operationen:**
  - `add(x)`:
    - Merken, wie viele Elemente bereits im Array
    - Falls noch Platz, rechts anhängen.
    - Sonst: `Exception`!
  - `indexOf(x)`:
    - Array von links nach rechts durchlaufen.
  - `remove(i)`:
    - Element aus Array löschen.
    - Alle Elements rechts davon um 1 nach links schieben. (**teuer!**)

Operation	**# Vergleiche (Worst Case)**
`add(x)`	$O(1)$
`indexOf(x)`	$O(n)$
`remove(i)`	$O(1)$ Aber: $\boldsymbol{O(n)}$ Kopier-operationen

ADT Liste als Array in Java (ohne "Resize")

```java
public class MyArrayList<Item> {

   private Item[] items;  // array to store items
   private int n;         // current number of
elements

   public MyArrayList(int capacity) {
      items = (Item []) new Object[capacity];
      n = 0;
   }

   // add element to arrayList
   public void add(Item x) throws Exception {
     if (n == items.length) {
         throw new Exception("No space left");
     }
     items[n] = x;
     n++;
   }

   // get position of element
   public int indexOf(Item x) {
      // TODO
   }

   // remove element with index i
   public Item remove(int index) {
      // TODO
   }

   public int size() {
      return n;
   }

   public boolean isEmpty() {
      return n > 0 ? true : false;
   }
}
```

Quellcode: MyArrayListWithoutResizing.java

- Idealerweise ist maximale Anzahl der Elemente beim Erzeugen bekannt.
  - Ansonsten: Dynamisches Vergrößern ("Resize")
  - Siehe übernächste Folie
- Java Standard Library („Collection Framework“): ***ArrayList***
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList

Exkurs: Generics

- Generische Typen erlauben *generische Datenstrukturen* (Flexibilität, Wiederverwendbarkeit) **und** *gleichzeitige Typprüfung durch den Compiler*.
  - Details: OOP und Fortgeschrittene Programmierkonzepte
  - Wichtige Anwendung: Java Collections

- Java erlaubt ***keine Arrays von parametrisierten Typen***
  - ~~`Item[] items = new Item[capacity]`~~
  - *"Generic array creation is disallowed in java"*

- ***Abhilfe mit Typecast***
  - `Item[] items = (Item []) new Object[capacity]`
  - Java Collections lösen das ähnlich.
  - Compilerwarnungen ggfs. ignorieren.

```
public class MyArrayList<Item> {

   private Item[] items;
   private int n;

   public MyArrayList(int capacity) {
      items = (Item []) new Item[capacity];
      n = 0;
   }
}
```

```
public class MyArrayList<Item> {

   private Item[] items;
   private int n;

   public MyArrayList(int capacity) {
      items = (Item []) new Object[capacity];
      n = 0;
   }
}
```

ADT Liste als Array: Resize

- Was tut man, wenn nicht von Anfang an bekannt ist, wie viele Elemente das Array später einmal speichern soll?
- ***Lösung:*** Dynamisches Vergrößern
- ***Nachteil:*** Teuer

```
// resize the underlying array holding the elements
private void resize(int capacity) {
   Item[] temp = (Item[]) new Object[capacity];
   for (int i = 0; i < n; i++)
      temp[i] = items[i];
   items = temp;
}

// add element to arrayList
public void add(Item x) throws Exception {
   if (n == items.length) {
      //throw new Exception("No space left");
      resize(2 * items.length);
   }
   items[n] = x;
   n++;
}
```

Laufzeit von `resize`?

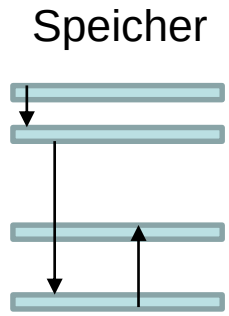
$O(n)$

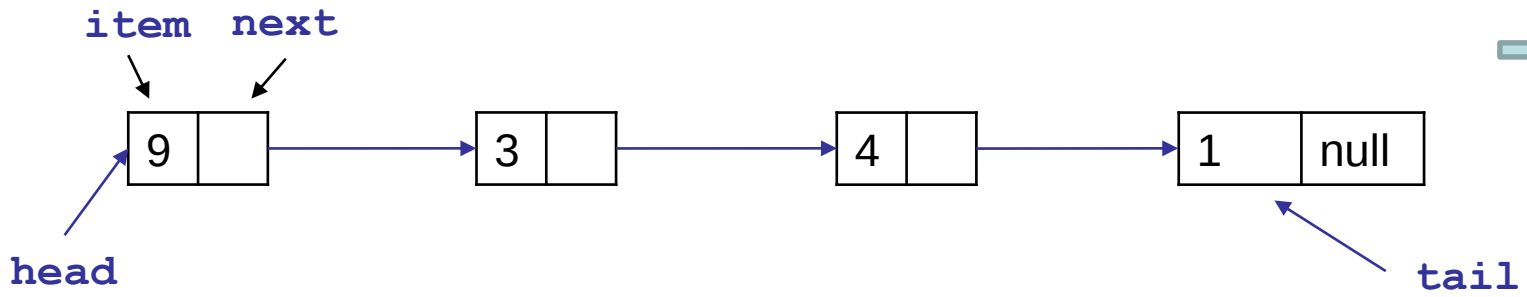
ADT Liste als verkettete Liste

- **Verkettete Liste**
  - Jedes Knoten speichert "Zeiger" auf nächsten (und vorherigen) Knoten.
  - **`head und tail:`** "Zeiger" auf ersten bzw. letzten Knoten
  - Java Collections: `LinkedList`!

- **Aufbau eines Knotens**
  - **`item`**: Nutzdatum, das was man eigentlich speichern möchte.
  - **`next:`** Zeiger auf nächstes Element
  - **`prev:`** Zeiger auf vorheriges Element (optional)

- **Optionen der Verkettung**
  - *Einfach verkettet:* Jedes Element speichert nur Nachfolger.
  - *Doppelt verkettet**:*** Jedes Element speichert Vorgänger ***und*** Nachfolger.
  - *Ringliste*: Letztes Element zeigt auf erstes Element.





Beispiel: Doppelt verkettete Liste

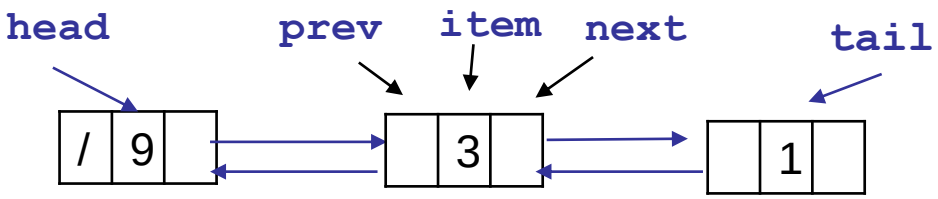
```
public class MyLinkedList<Item> {
    // inner class
    private class Node {
        Item item;
        Node next;  // next node
        Node prev;  // previous node
    }
    private Node head;  // first node
    private Node tail;  // last node
    private int n;      // current number of elements

    // add element to the end
    public void add(Item x)  {
        Node temp = new Node();
        temp.item = x;
        temp.prev = tail;
        temp.next = null;

        if (n == 0)
           head = temp;
        else
           tail.next = temp;

        tail = temp;
        n++;
    }
. . .
}
```

Quellcode: MyLinkedList.java



Operation	Laufzeit (Worst Case)
add(x)	$\Theta(1)$
indexOf(x)	$\Theta(n)$ (kein Verschieben von Elementen)
remove(x)	$\Theta(1)$ (Kein Verschieben der Elemente, nur Umbiegen der Zeiger)

Publikums-Joker: Array vs. Verkettete Liste

Welche der folgenden Aussagen ist **falsch**?

A. Das Einfügen von Elementen an beliebiger Position ist bei verketteten Listen einfacher als bei Arrays.

B. Der Zugriff auf ein Element an beliebiger Position ist einfacher, falls die Liste als Array implementiert ist.

C. Verkettete Listen sind immer dann besser geeignet als Arrays, falls die Anzahl der einzufügenden Elemente zwar bekannt aber sehr groß ist.

D. Arrays sind oft effizienter als verkettete Listen wegen der räumlichen Lokalität von benachbarten Elementen (Cache Lokalität).

Durchlaufen von Listen, Iterator

- Der ADT Liste benötigt häufig eine Operation zum ***Durchlaufen aller Elemente.***

- In Java wird das mit einem ***Iterator*** gelöst (siehe OOP)
  - Der Benutzer sieht nur den Iterator.
  - Wie das Iterieren gelöst ist und mit welcher Datenstruktur (Verkettete Liste, Array), kann dem Benutzer egal sein.

```
public class MyLinkedList<Item> implements
Iterable<Item> {
    // inner class
    private class Node<Item> {
        Item item;
        Node next;  // next node
        Node prev;  // previous node
    }
    private Node head;  // first node
    private Node tail;  // last node
    private int n;        // current number
of elements

    // returns an iterator that iterates
over the items
    public Iterator<Item> iterator()  {
        return new ListIterator(head);
    }
    . . .
```

```
// inner class of MyLinkedList
private class ListIterator implements Iterator<Item> {
    private Node<Item> current;

    public ListIterator(Node<Item> first) {
        current = first;
    }

    public boolean hasNext()  { return current != null;}

    public Item next() {
        if (!hasNext()) throw new NoSuchElementException();
        Item item = current.item;
        current = current.next;
        return item;
    }
}
```

Inhalt

- Datenstrukturen
- Listen
- **Stacks und Queues**
- Mengen

ADTs: Stack und Queue

- Spezialfälle von ADT "Liste"
- Einfügen und Löschen nur an **fester Position** erlaubt.
  - **Stack** (dt. Stapel/Keller): *Last in, first out* → Lösche zuletzt eingefügtes Element
  - **Queue** (dt. "Warteschlange"): *First in, first-out* → Lösche ältestes Element

Operationen eines Stacks

- `push(x)`
  - Füge ein Element x hinzu
- `pop()`
  - Entferne das zuletzt eingefügte („oberste“) Element
- `peek()`
  - Schaue oberstes Element an, ohne es zu entfernen.
- `empty()`
  - Ist Stack leer?

Operationen einer Queue

- `enqueue(x)`
  - Reihe ein Element x in die Schlange ein.
- `dequeue()`
  - Entferne das älteste Element aus der Schlange.
- `empty()`
  - Ist Queue leer?

Animationen: https://visualgo.net/en/list

Anwendung von Stacks und Queues

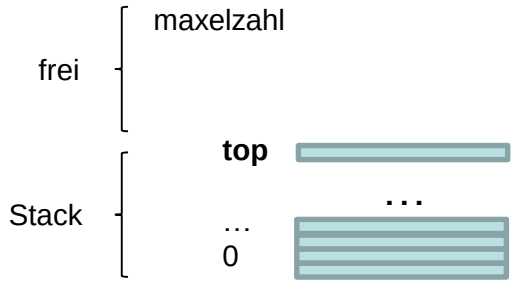
- **Stacks**
  - Auswerten von Rechenausdrücken
  - Bau von Compilern
  - Aufruf von Unterprogrammen
  - Tiefensuche in Graphen

- **Queues**
  - Druckaufträge, Druckerwarteschlange
  - Verwaltung von Prozessaufrufen im Betriebssystem
  - Hardwareschnittstellen zur seriellen Ein-/Ausgabe
    - Bsp.: Pufferspeicher in Tastatur
  - Priority Queues (dt. „Vorrangwarteschlangen“):
    - Elemente mit höherer Priorität werden zuerst bedient.
    - Spezialisierung der ADT „Queue“ (siehe späterer Teil der Vorlesung)

Umsetzung ADT Stack und Queue

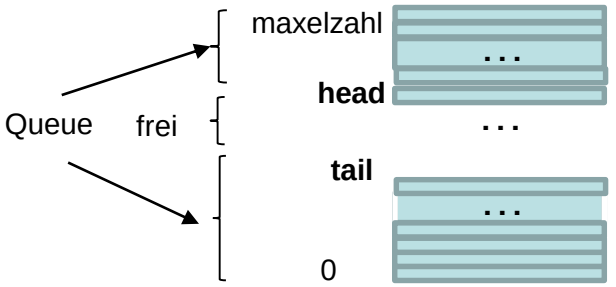
- **Stack**
  - *Verkettete Liste*: Elemente z.B. immer am Ende einfügen und entfernen
  - *Array*: Bestimmte Größe vorreservieren, Elemente immer „hinten“ einfügen und entfernen.

Stack als Array



- **Queue**
  - *Verkettete Liste*: Elemente hinten (`tail`) einfügen und vorne (`head`) entfernen.
  - *Array*: Verwendeter Adressbereich verschiebt sich permanent → Überlauf / Ringpuffer!

Queue als Array



`head` zeigt auf das nächste zu entfernende Element!
`tail` zeigt auf nächsten Index, an dem eingefügt werden muss

- **Ziel**
  - Alle Operationen soll in *konstanter Zeit* $O(1)$ ausführbar sein

Implementierung eines Stacks über ein Array

```
public class MyArrayStack<Item> {
    private Item[] items;  // stack entries;
    private int n;         // stack size;

    public MyArrayStack(int capacity) {
        items = (Item[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return n == 0;
    }
    public void push(Item item) {
        items[n++] = item;
    }
    public Item pop() {
        Item temp = items[--n];
        return temp;
    }
    public Item peek() {
        return items[n-1];
    }
}
```

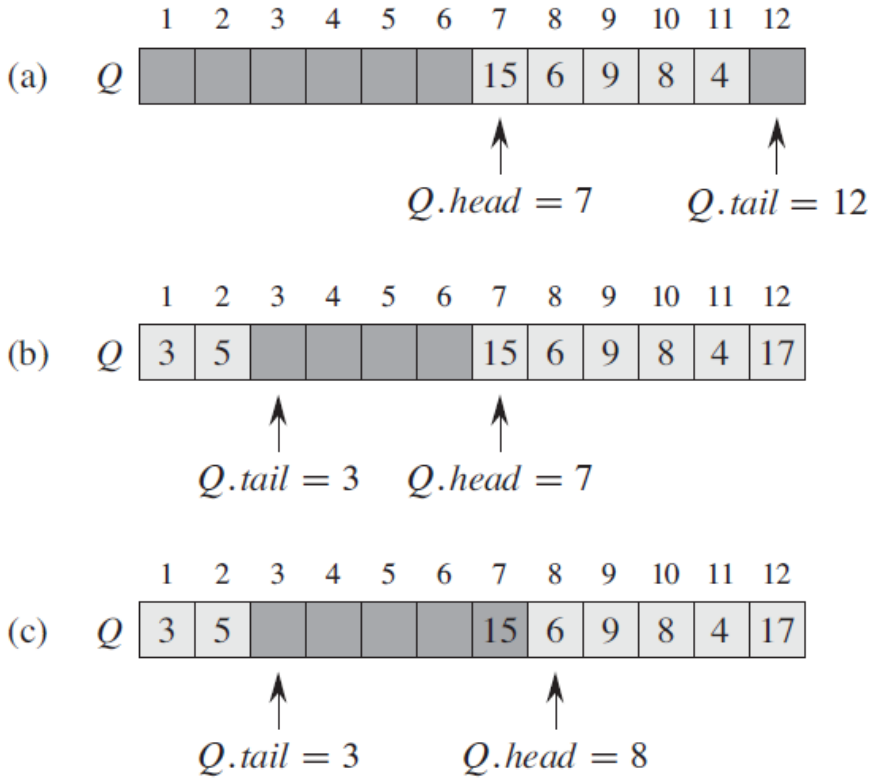
Quellcode: MyArrayStack.java

Operation	Laufzeit (Worst Case)
isEmpty()	$\Theta(?)$
push(x)	$\Theta(?)$
pop()	$\Theta(?)$

- Stacks in Java:
  - Array-basiert: `Stack<E>`
  - Über verkettete Liste: `LinkedList<E>` bzw. `ArrayList<E>`
- Animationen
  - https://www.cs.usfca.edu/~galles/visualization/StackArray.html
  - https://www.cs.usfca.edu/~galles/visualization/StackLL.html

Implementierung einer Queue durch ein Array

- **`tail`**
  - Index, an dem nächstes Element ***eingefügt*** werden muss.
- **`head`**
  - Index an dem nächstes Element ***entfernt*** werden muss.



Die Queue hat zunächst 5 Elemente!

Queue nach Einfügen von 17, 3, und 5!

Queue nach einmaligen DEQUEUE.

Achtung: In dieser Grafik ist der erste Arrayindex 1 und nicht 0

Implementierung einer Queue durch ein Array

```
public class MyArrayQueue<Item>
{
    private Item[] items;
    private int n;
    private int head;
    private int tail;

    // create an empty queue
with 10 elements
    public MyArrayQueue() {
        items = (Item[])
          new Object[10];
        n = 0;
        head = 0;
        tail = 0;
    }
   . . .
```

```
public void enqueue(Item item) {
    // double size of array if necessary
    if (n == items.length) {
        resize(2 * items.length);
    }
    items[tail++] = item;         // add item
    if (tail == items.length) {
        tail = 0;                 // wrap-around
    }
    n++;
}
public Item dequeue() {
    if (isEmpty()) throw new NoSuchElementException(" ");
    Item item = items[head];
    items[head] = null;           // to avoid loitering
    n--;
    head++;
    if (head == items.length) head = 0; // wrap-around
    return item;
}
```

MyArrayQueue.java

Laufzeiten?

- Array-basierte Queue in Java: `ArrayDeque<E>`
- Animation
  - https://www.cs.usfca.edu/~galles/visualization/QueueArray.html
  - https://www.cs.usfca.edu/~galles/visualization/QueueLL.html

Publikums-Joker: Array vs. Verkettete Liste

Welche der folgenden Aussagen ist ***falsch***?

A. Sowohl eine Queue als auch ein Stack kann in Java mittels der Klasse `LinkedList` implementiert werden.

B. Das Löschen eines Elements ist bzgl. der asymptotischen Worst Case Laufzeit schneller wenn die Liste doppelt verkettet ist als wenn sie einfach verkettet ist.

C. Eine Queue kann sowohl über ein Array als auch über eine doppelt verkettete Liste implementiert werden.

D. Druckerjobs werden mit einer Queue verwaltet.

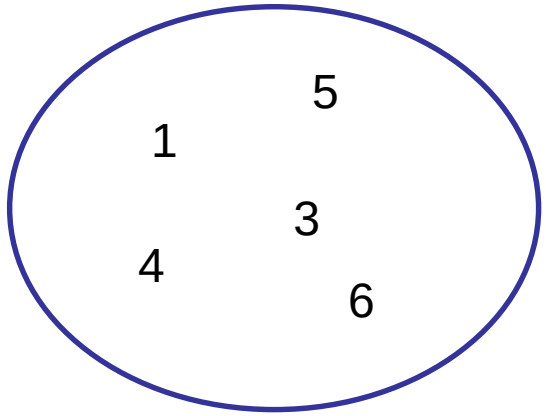
Inhalt

- Datenstrukturen
- Lineare Listen
- Stacks und Queues
- **Mengen**

ADT: Menge

- „Mathematische“ Menge
- Keine Ordnung, also kein 1., 2., 3. Element, …
- Jedes Element darf nur einmal enthalten sein.

- ***Typische Operationen***
  - Hinzufügen: `add`
  - Entfernen: `remove`
  - Ist Element enthalten? `contains`
  - Ist die Menge leer? `isEmpty`
  - Durchlaufen aller Elemente, Iterator



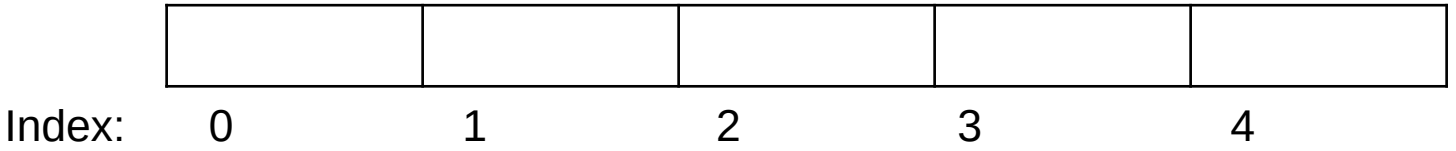
- ***Implementierung***
  - Bäume, Hashtabelle (siehe später)
  - Bitvektor, Array

- **Java-Klassen:** `TreeSet, HashSet, BitSet`

ADT Menge über Bitvektor, Array

- Menge *M* wird beschrieben durch (boolesches) Array *M*[0..*n*-1]
  - *M*[*i*] = 1 falls *i* in *M*
  - *M*[*i*] = 0 falls *i* nicht in *M*

- ***Übung:*** Wie sieht Bitvektor aus, falls $M = \{1,3,4\}$ und bekannt ist, dass alle Werte kleiner als 5 sind?

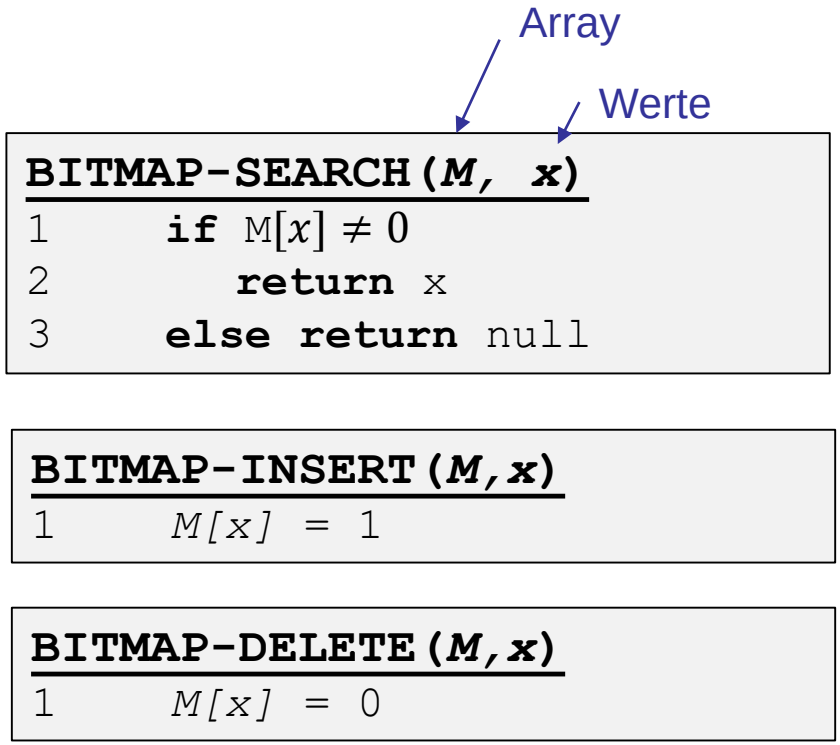


- Eine Menge sollte nur dann als Bitvektor implementiert werden, falls
  - jedes Element eineindeutig einem Integer (=Index) zuzuordnen ist.
  - der Wertebereich der Indizes vorher bekannt ist.

- Java-Klasse: Bitset
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/BitSet.html

ADT Menge als Bitvektor

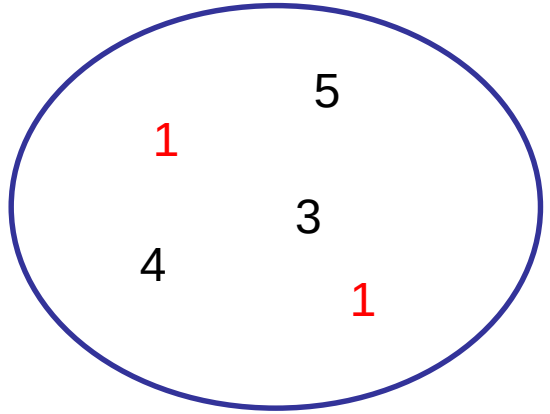
- ***Laufzeit***
  - Alle 3 Operationen sind konstant: $O(1)$
- ***Speicherplatz***
  - Günstig, da jeder Schlüssel zum Abspeichern nur 1 Bit benötigt.
- ***Nachteile***
  - Siehe Vorgängerfolie



Pseudocode

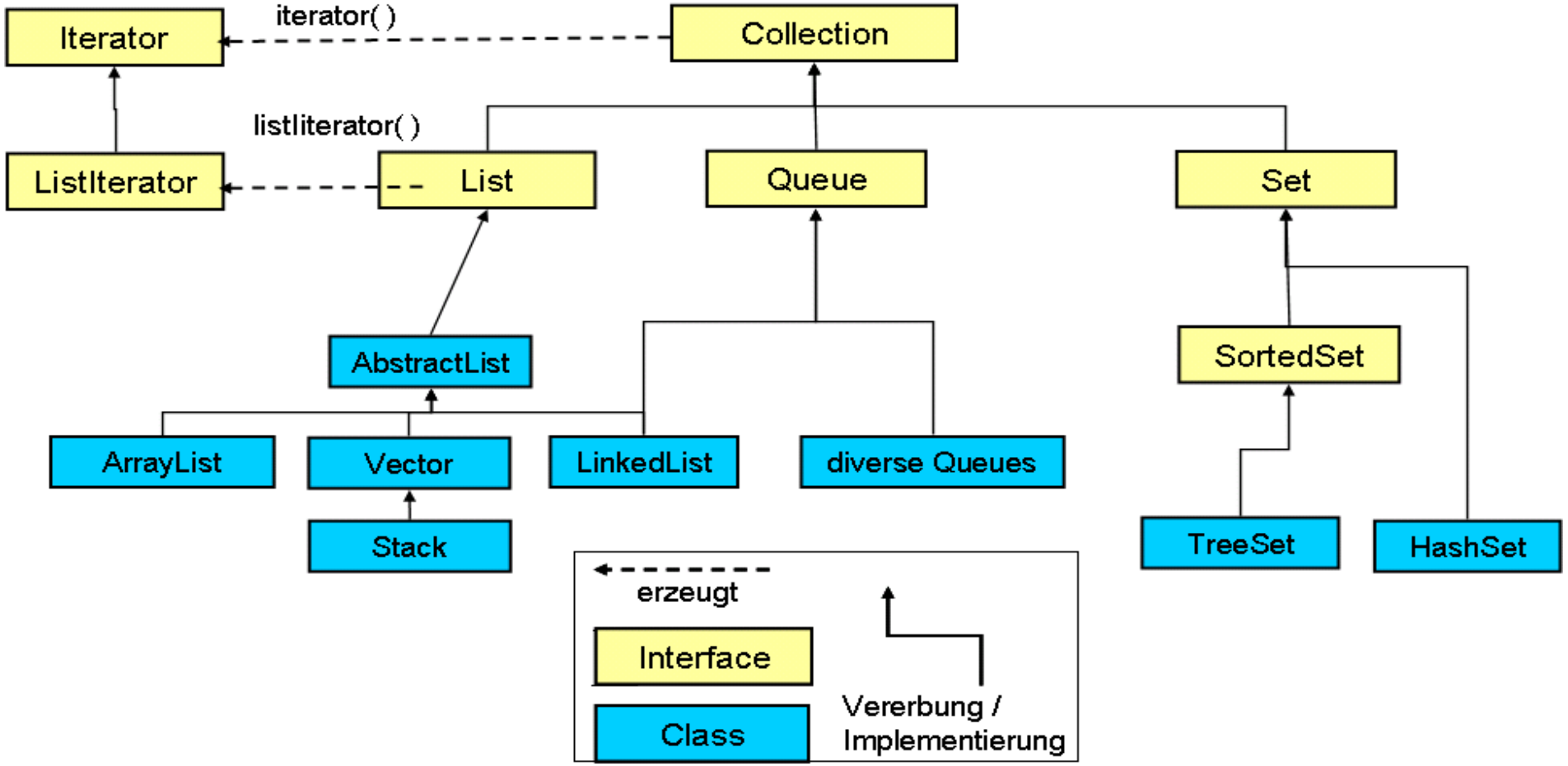
ADT Multimenge (engl. "Bag" / "Multiset")

- Wie "normale" Menge, aber Elemente dürfen doppelt vorkommen.



- Preisfrage: Was verwendet man in Java, um eine Multimenge zu implementieren?

Datenstrukturen im Java Collection Framework



Zusammenfassung

- ADTs
  - Liste
  - Stack
  - Queue
  - Menge
  - Multimenge

implementiert durch

- Datenstrukturen
  - Array, [Bitvektor]
  - Verkettete Listen

- **Animationen**
  - https://www.cs.usfca.edu/~galles/visualization/

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 1.2.3, 5. Auflage, Spektrum Akademischer Verlag, 2012. (Karazuba-Algorithmus, eBook!)

[3] https://de.wikipedia.org/wiki/Quipu (abgerufen am 16.10.2016)

[4] https://s3.amazonaws.com/lowres.cartoonstock.com/business-commerce-pc-neat-tidy_desks-desks-file-cwln557_low.jpg (abgerufen am 16.10.2016)

[5] https://s3.amazonaws.com/lowres.cartoonstock.com/business-commerce-pc-neat-tidy_desks-desks-file-cwln557_low.jpg