Kapitel 1: Grundlagen

Algorithmen Basics
Datenstrukturen Basics

DBS Prof. Dr. Thomas Seidl

Probleme in der Informatik

- Ein *Problem* (im Sinne der Informatik):
  - Enthält eine Beschreibung der Eingabe
  - Enthält eine davon abhängige Ausgabe
  - Gibt keinen Übergang von Eingabe zu Ausgabe an

Eingabe		Ausgabe
$x \in \mathbb{R}^+$	?	$y = \sqrt{x} \in \mathbb{R}$

- Beispiele:
  - Sortiere eine Menge von Wörtern
  - Berechne die Quadratwurzel von x
  - Finde den kürzesten Pfad zwischen 2 Orten

Probleminstanzen

- Eine <u>*Probleminstanz*</u> ist eine konkrete Eingabebelegung, für die die entsprechende Ausgabe gewünscht ist.

- Beispiele für Probleminstanzen:
  - Sortiere folgende Wörter alphabetisch: {Haus, Auto, Baum, Tier, Mensch}
  - Berechne $\mathrm{x} = \sqrt{204}$
  - Was ist der kürzeste Weg vom Hörsaal in die Mensa?

Zentraler Begriff Algorithmus

„Ein *Algorithmus* ist
eine **endliche Sequenz** von Handlungsvorschriften,
die eine **Eingabe** in eine **Ausgabe** transformiert.“

Cormen et al., 2009

Die Bezeichnung „Algorithmus“ leitet sich vom arabischen Mathematiker „al-Chwarizmi“ (um 800 n.Chr.) ab.

Anforderungen an Algorithmen

- Spezifizierung der Eingabe/Ausgabe:
  - Anzahl und Typen aller Elemente ist definiert.
- Eindeutigkeit:
  - Jeder Einzelschritt ist klar definiert und ausführbar.
  - Die Reihenfolge der Einzelschritte ist festgelegt.
- Endlichkeit:
  - Die Notation hat eine endliche Länge.

Eingabe
$x, \epsilon \in \mathbb{R}^+$

Algorithmus →

Ausgabe
$y = \sqrt{x} \in \mathbb{R}$

$$y = \frac{x+1}{2}$$

Wiederhole bis $|y^2 - x| \leq \varepsilon$:

$$y = \frac{1}{2}\left(y + \frac{x}{y}\right)$$

Beispiel SummeBis(n) in natürlicher Sprache

- Problem:
  - Für ein gegebenes $n \in \mathbb{N}$,
    berechne die Summe $1 + 2 + \cdots + n$

- Natürliche Sprache:
  - Initialisiere eine Variable **summe** mit Wert 0. Durchlaufe die Zahlen von 1 bis n mit einer weiteren Variable **zähler**. Addiere **zähler** jeweils zu **summe**. Gib nach dem Durchlauf den Text „Die Summe ist: “ und den Wert von **summe** aus.

Beispiel SummeBis(n) in Pseudocode

- Problem:
  - Für ein gegebenes $n \in \mathbb{N}$,
    berechne die Summe $1 + 2 + \cdots + n$

- Pseudocode:

  Setze **summe** = 0

  Setze **zähler** = 1

  Solange **zähler** $\leq n$

      setze **summe** = **summe** + **zähler**

      erhöhe **zähler** um 1

  Gib aus: „Die Summe ist: “ und **summe**

Beispiel SummeBis(n) in Javacode

- Problem:
  - Für ein gegebenes $n \in \mathbb{N}$,
    berechne die Summe $1 + 2 + \cdots + n$

- Java:

```
public static int SummeBis(n) {
  int sum = 0;
  for (int i = 1; i <= n; ++i)
    sum += i;
  System.out.println („Die Summe ist: “ + sum);
  return sum;
}
```

Beispiel SummeBis(n) in Python

- Problem:
  - Für ein gegebenes $n \in \mathbb{N}$,
    berechne die Summe $1 + 2 + \cdots + n$

- Python:

```
def SummeBis(n):
  sum = 0
  for i in range(n):
    sum += i
  print(„Die Summe ist: “ + str(sum))
  return sum
```

Einige Eigenschaften von Algorithmen

- Allgemeinheit:
  - Lösung für Problemklasse, nicht für Einzelaufgabe
- Determiniertheit:
  - Für die gleiche Eingabe wird stets die gleiche Ausgabe berechnet (aber andere Zwischenzustände möglich).
- Determinismus:
  - Für die gleiche Eingabe ist die Ausführung stets identisch.
  - Bsp. nichtdeterministisch: $abs(x) = -x$ falls $x \leq 0, x$ falls $x \geq 0$
- Terminierung:
  - Der Algorithmus läuft für jede Eingabe nur endlich lange
- (partielle) Korrektheit:
  - Algorithmus berechnet stets die spezifizierte Ausgabe (falls er terminiert)
- Effizienz:
  - Sparsamkeit im Ressourcenverbrauch (Zeit, Speicher, Energie, …)

Lernziele der Vorlesung (Algorithmen)

Nach dieser Vorlesung können Sie:

- Viele Probleme analysieren und strukturieren
- Für einige Problemklassen den passenden Algorithmus auswählen
- Algorithmen auf Probleminstanzen anwenden
- Den Rechenaufwand eines Algorithmus quantifizieren
- Die Effizienz und Anwendbarkeit mehrerer Algorithmen miteinander vergleichen

Zentraler Begriff Datenstruktur

„Eine *Datenstruktur* ist ein Weg, Daten zu **speichern** und zu **organisieren**, so dass **Zugriffe** und **Modifikationen** darauf ermöglicht werden."

Cormen et al., 2009

Datenstrukturen

- Datenstrukturen
  - Organisationsformen für Daten
  - Funktionale Sicht: Containerobjekte mit Operationen, lassen sich als abstrakte Datentypen beschreiben.
  - Beinhalten Strukturbestandteile und Nutzerdaten (Payload)
  - Können gleichförmig oder heterogen strukturiert sein
  - Anforderungen:
    - Statisch oder dynamisch bestimmte Größe
    - Transiente oder persistente Speicherung

- Betrachtete Beispiele
  - Sequenzen: Arrays, Listen, Kellerspeicher, Warteschlangen
  - Multidimensional: Matrizen
  - Topologische Strukturen: Bäume, Graphen, Netzwerke

Lernziele der Vorlesung (Datenstrukturen)

Nach dieser Vorlesung können Sie:

- Grundlegende Datenstrukturen erkennen.
- Zugehörige Basisoperationen auf Strukturen anwenden.
- Die Laufzeiten eines Algorithmus mit verschiedenen Datenstrukturen abschätzen.
- Eine geeignete Datenstruktur für eine Lösungsstrategie auswählen.
- Ähnliche Datenstrukturen miteinander vergleichen.

Datentypen

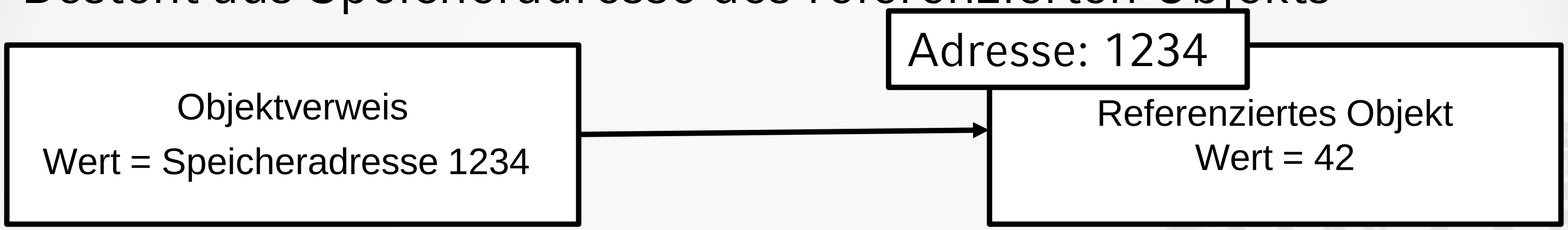
- Definition: Menge von Werten und Operationen auf diesen Werten

- Elementare (atomare) Datentypen: (Java)
  - Ganze Zahlen: byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
  - Binärer Wahrheitswert (true oder false): boolean
  - Zeichen: char (16-bit)
  - Fließkommazahlen: float (32-bit), double (64-bit)

- Zusammengesetzte Typen:
  - String: Zeichenkette
  - Record: Datensatz (in Java nicht explizit; als Objekt o.ä.)
  - Set: Menge (in Java vordefiniert, inklusive Methoden zum Sortieren etc.)
  - Array: Reihung fester Länge von gleichartigen Daten

Objektverweise als Zeiger (Pointer)

- In Java nicht explizit
- Referenz auf ein anderes Objekt
- Besteht aus Speicheradresse des referenzierten Objekts

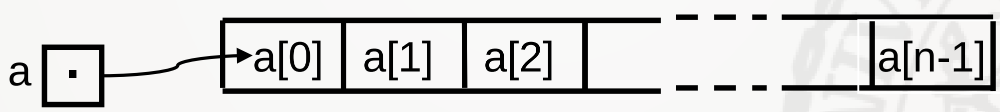


- Für dynamische Datenstrukturen: Speicher erst bei Bedarf
- In einigen Programmiersprachen explizite Speicherfreigabe
- Java hat „garbage collection“:
  Falls keine Referenz mehr vorhanden ist, wird der Speicher freigegeben

Zusammengesetzte Typen: Arrays

- Array: Reihung (Feld) fester Länge von Daten gleichen Typs
  - z.B. a[i] bedeutet Zugriff auf das $(i+1)$-te Element eines Arrays a[]
  - Erlaubt effizienten Zugriff auf Elemente: konstanter Aufwand
  - Wichtig: Array-Grenzen beachten!

  - Referenz-Typ: Verweis auf (Adresse der) Daten



  - Vorsicht: Array a beginnt in Java bei 0 und geht bis a.length – 1 (Häufige Fehlerquelle)

Beispiel: Sieb des Eratosthenes

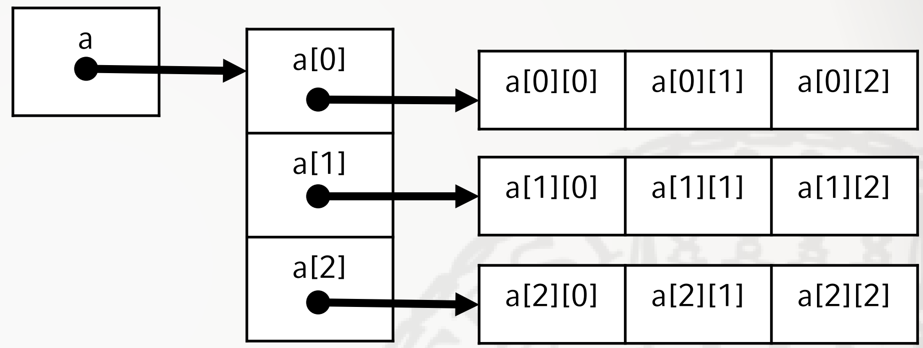
- Eratosthenes (hellenischer Gelehrter, ca. 276–195 v. Chr.)
  - Problem: Suche alle Primzahlen kleiner n
  - Idee: Benutze Array a mit Codierung $a[i] = \begin{cases} 1 & \text{falls } i \text{ prim ist} \\ 0 & \text{sonst} \end{cases}$
  - Algorithmus:
    - Initialisiere Werte $a[1]$ bis a$[n]$ mit 1 (d.h. „prim")
    - Setze Vielfache sukzessive auf 0 (d.h. „nicht-prim")
    - Arrayeinträge sind nun 1, falls ihre Indizes prim sind
  - Beispiel für n = 25: *Frage: bei welchem i darf man stoppen?*

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2			0		0		0		0		0		0		0		0		0		0		0	
3								0						0						0				
5																								0

Mehrdimensionale Arrays

- Zweidimensionale Arrays (= Matrizen) sind Arrays von Arrays

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]



- Deklaration

```
int [][] a = new int [4] [3];      // keine Initialisierung
int [][] m = {{1,2,3},{4,5,6}};    // Initialisierung mit Konstanten
```

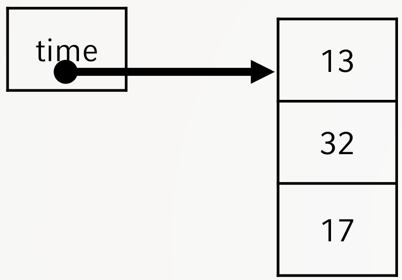
- Höhere Dimensionen

```
int [][][] q = new int [2][2][2];  // 3D: Quader, Tensor
```

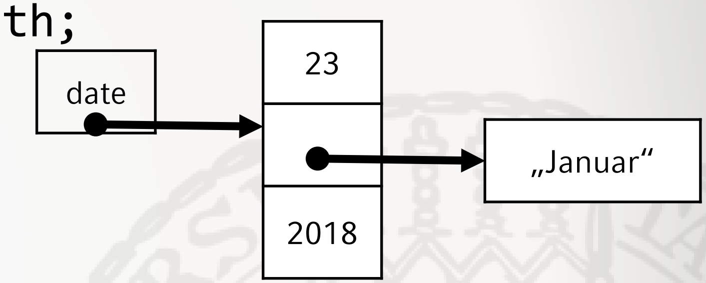
Benutzerdefinierte Datentypen: Klassen

- Zusammenfassung verschiedener Attribute zu einem Objekt

```
class Time {
    int h, m, s;
}
```




```
class Date {
  int day;
  String month;
  int year;
}
```



- Beispiel: Rückgabe mehrerer Funktionsergebnisse auf einmal
  - Java erlaubt nur einen einzigen Rückgabewert
  - Lösung: Rückgabe eines komplexen Ergebnisobjekts

```
static Time convert (int sec) {
    Time t = new Time();
    t.h = sec / 3600; t.m = (sec % 3600) / 60; t.s = sec % 60;
    return t;
}
```

Heterogene vs. homogene Datensätze

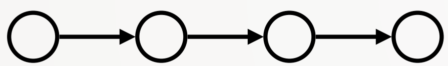
- **Klassen** eignen sich zur Speicherung von **heterogenen** Datentypen
  - Bestehen im allgemeinen aus verschiedenartigen Elementen:
    ```
    class c {String s; int i;}
    ```
  - Jedes Element hat einen eigenen Namen: `c.s, c.i`
  - Anzahl der Elemente wird statisch bei der Deklaration der Klasse festgelegt.

- **Arrays** ermöglichen schnellen Zugriff auf **homogene** Daten
  - Bestehen immer aus mehreren gleichartigen Elementen: `int[]`
  - Elemente haben keine eigenen Namen, sondern werden über Indizes angesprochen: `a[i]`
  - Anzahl der Elemente wird dynamisch bei der Erzeugung des Arrays festgelegt:
    ```
    new int[n]
    ```

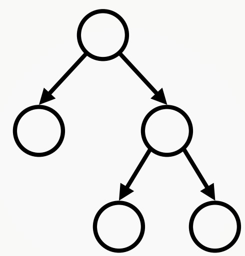
- **Frage:** Welche Kombinationen von Klassen und Arrays sind möglich?

Dynamische Datenstrukturen

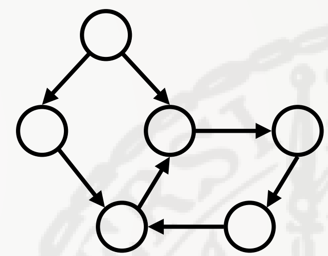
- Motivation
  - Länge eines Arrays ist nach der Erzeugung festgelegt
  - hilfreich wären unbeschränkt große Datenstrukturen
  - Lösungsidee: Verkettung einzelner Objekte zu größeren Strukturen
- Beispiele



Liste



Baum

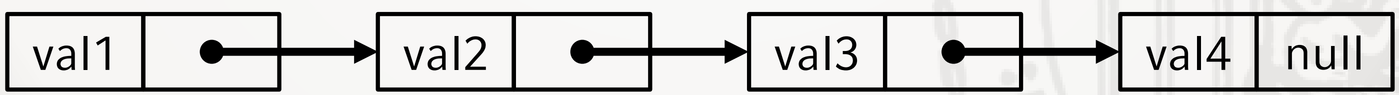


Graph

- Charakterisierung
  - Knoten werden zur Laufzeit (dynamisch) erzeugt und verkettet
  - Strukturen können dynamisch wachsen und schrumpfen
  - Größe einer Struktur ist nur durch verfügbaren Speicherplatz beschränkt; muss nicht im vorhinein bestimmt werden.

Listen

- Rekursive Definition
  - list = val ∘ list = val ∘ val ∘ list = val ∘ val ∘ val ∘ list = ???
  - Lösung: Zulassen einer leeren Liste hilft
  - Also („|“ steht für „oder“): list = null | val ∘ list
- Funktionale Zerlegung einer Liste
  - Liste L = head(L) ∘ tail(L) = value ∘ next
  - Beispiel: {1,2,3,4} = {1} ∘ {2,3,4}
- Beispiel:

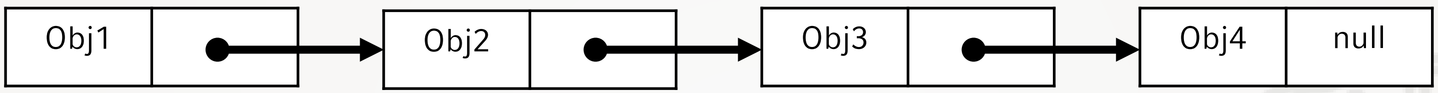


Listen in Python

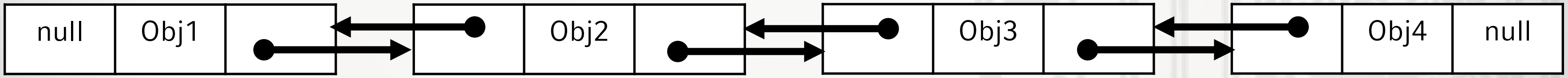
```
# Definition
class List:
  # Constructor
  __init__(self, value = None, next = None):
    self.value = value    # „payload“
    self.next = next      # next darf null sein.

  # Print
  def show(self):
    print(self.value, end = " ")
    if self.next != None:
      self.next.show()

# Example call
l = List(1,List(2,List(3)))
l.show()                          # Out: 1 2 3
```

Listen – Verkettung

- Einfach verkettete Liste
  - Jeder Knoten enthält Verweis auf nächsten Knoten

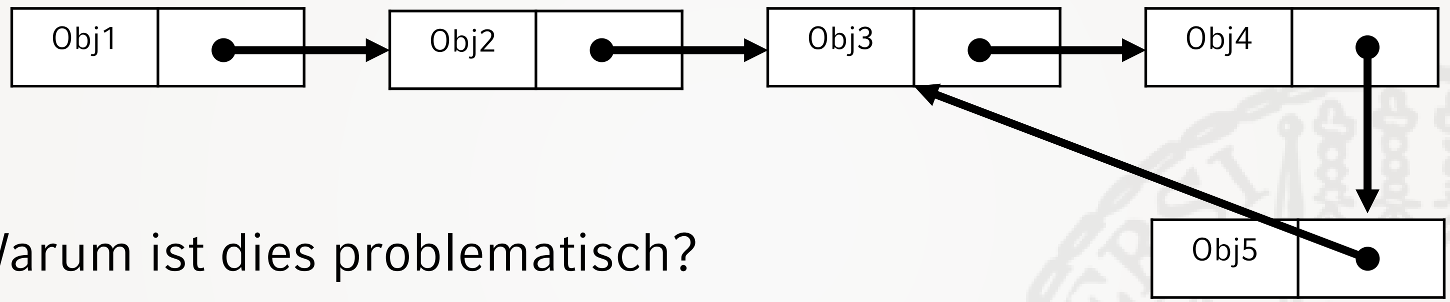


- Doppelt verkettete Liste
  - Jeder Knoten enthält zusätzlich Verweis auf vorherigen Knoten



Zyklenfreiheit

- Implementierungen von Liste sollten keine Konstruktion von Zyklen (Kreisen) innerhalb der Liste erlauben



- Warum ist dies problematisch?
  - Was ist die Länge dieser Liste?
  - Wo ist das Ende?
  - Wie füge ich weitere Elemente hinten an?

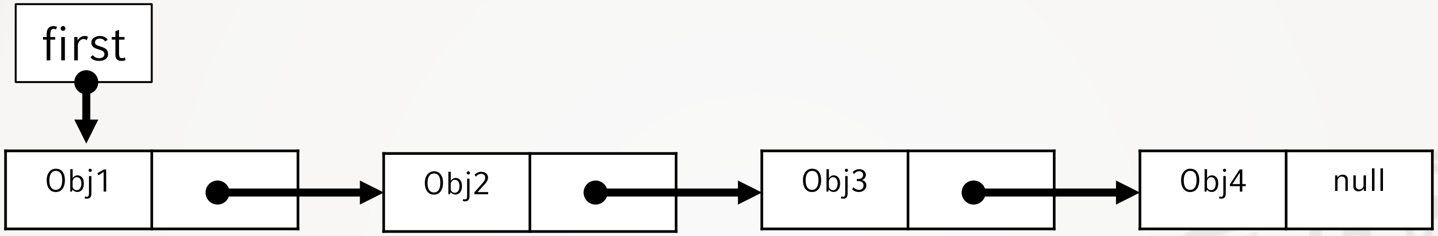
- Um die Liste "sicherer" gegen ungewollte Manipulation zu machen, kapseln wir die Knoten in eine eigene Klasse.

Listen in Python mit Wrapper

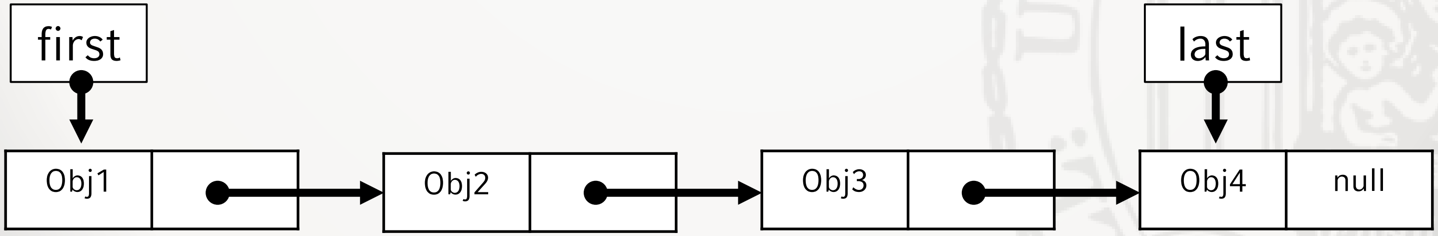
```
# Recursive Entries
class _Entry:
  def __init__(self, value = None, next = None):
    self.value = value
    self.next = next

# Wrapper
class List:
  def __init__(self, first = None):
    self.first = first
    self.last = first
    if self.last == None:
      return
    while self.last.next != None:
      self.last = self.last.next
```

Listen – Verankerung

- Einfach verankerte Liste
  - Liste enthält Zeiger auf erstes Element (seltener auf das letzte).



- Doppelt verankerte Liste
  - Es gibt sowohl Zeiger auf das erste und das letzte Element.



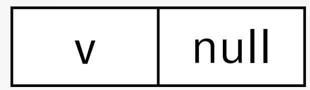
Funktionen in List-Wrapper

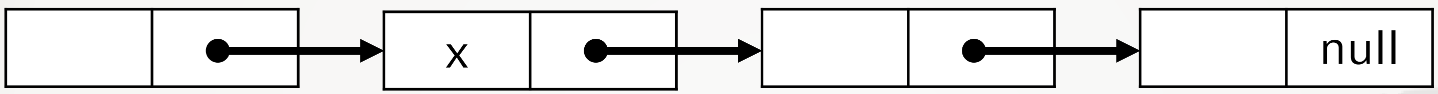
- Die Entry-Klasse enthält die Daten der Liste und stellt die Struktur sicher.
- Die List-Klasse definiert Funktionen zur Manipulation der Listenelemente.
  - Hinzufügen eines Elements am Listenende (append)
  - Hinzufügen eines Elements an Position (insert)
  - Entfernen eines Elements an Position (remove)
  - Entfernen des letzten Elements (pop)
  - Zugriff auf Element an Position (get)
  - Ausgabe aller Elemente (print)
  - Listeneigenschaften bestimmen wie Länge
  - Konkatenation mit anderer Liste
  - Evtl. Sortieren, Suchen
  - …

Listen – Einfügen

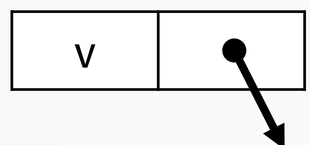
- Wert v nach Knoten x einfügen

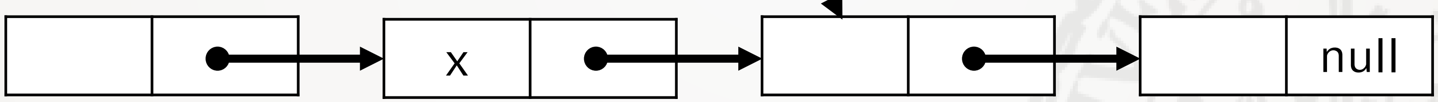
`e = _Entry(v)`



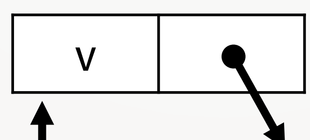


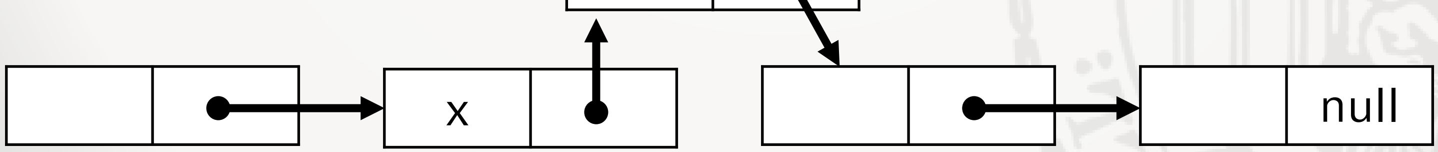
`e.next = x.next`





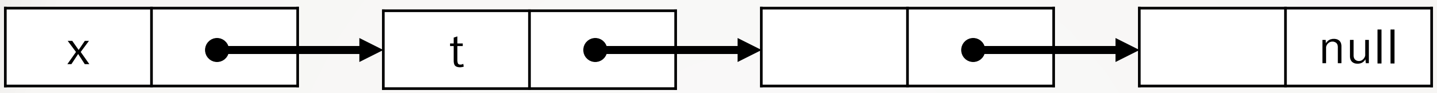
`x.next = e`



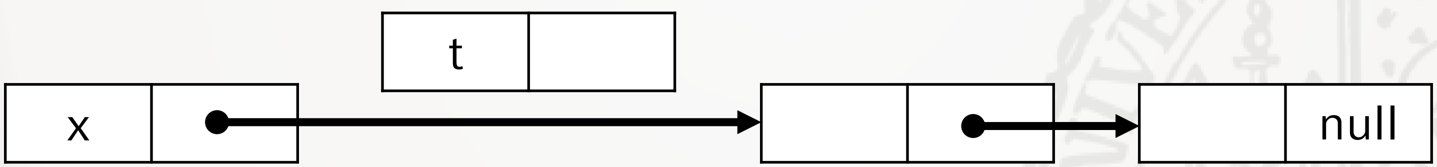


Listen – Löschen

- Knoten t nach Knoten x löschen




```
x.next = x.next.next
```



Abstrakte Datentypen

- Datenstruktur definiert durch auf ihr zugelassener Methoden
- Spezielle Implementierung nicht betrachtet
- Definition über:
  - Menge von Objekten
  - Methoden auf diesen Objekten $\rightarrow$ Syntax des Datentyps
  - Axiome $\rightarrow$ Semantik des Datentyps
- Top-down Software-Entwurf
- Spezifikation
  - Zuerst „was“ festlegen, noch nicht „wie“
    - Spezifikation vs. Implementierung
  - Klarere Darstellung von Programmkonzepten
- Abstraktion in Java:
  - Abstract class
  - Interface

Beispiel: Algebraische Spezifikation Boolean

- Wertebereich:
  - {true, false}

- Operationen:
  - NOT (Zeichen $\neg$): boolean $\rightarrow$ boolean
  - AND (Zeichen $\wedge$): boolean $\times$ boolean $\rightarrow$ boolean
  - OR (Zeichen $\vee$): boolean $\times$ boolean $\rightarrow$ boolean

- Axiome:
  - $\neg$ true = false; $\neg$ false = true;
  - x $\wedge$ true = x; x $\wedge$ false = false;
  - x $\vee$ true = true; x $\vee$ false = x;

a	b	$\neg$a	a$\wedge$b	a$\vee$b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Stack

- Stapel von Elementen („Kellerspeicher“)
- Wie Liste: sequentielle Ordnung, aber nur Zugriff auf erstes Element:

```
interface Stack {
  void push (Object);  // neues Element oben einfügen
  Object pop ();       // oberstes Element ausgeben und entfernen
  boolean isEmpty();
}
```

- Ein Stack folgt dem Prinzip LIFO: Last-in-first-out
- LIFO lässt sich formal fassen: Für alle Stack s und Object o gilt nach `s.push(o) immer s.pop() == o`

Algebraische Spezifikation Stack

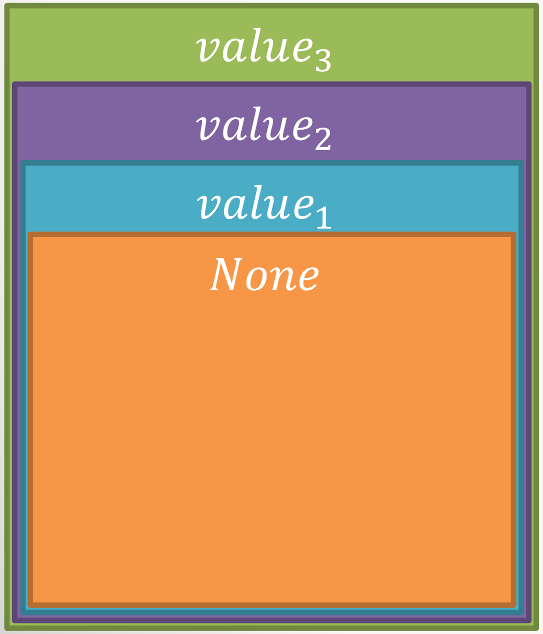
- Operationen:
  - Init: $\rightarrow$ Stack
  - isEmpty: Stack $\rightarrow$ Boolean
  - Push: Element $\times$ Stack $\rightarrow$ Stack
  - Pop: Stack $\rightarrow$ Element $\times$ Stack

- Axiome: Für alle Elemente x, Stack s gelten folgende Geleichungen:
  - Pop(Push(x,s)) = (x,s)
  - Push(Pop(s)) = s für isEmpty(s) = FALSE
  - Empty(Init) = TRUE
  - Empty(Push(x,s)) = FALSE

- Undefinierte Operationen erfordern Fehlerbehandlung
  - Beispiel: Pop (Init)

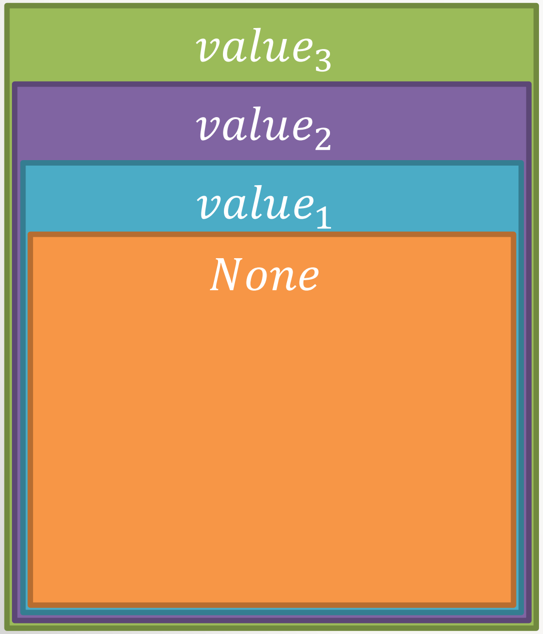
Stack

- Wir können hier eine rekursive Struktur verwenden:
$$stack = (v_n, (v_{n-1}, (v_{n-2}, (\dots (v_1, None) \dots))))$$
- Auf das äußerste Element kann direkt zugegriffen werden.
- Alle inneren Elemente erfordern das Auspacken der äußeren Elemente.
- Das Verarbeitungsprinzip nennt man „Last-In-First-Out“ (LIFO)



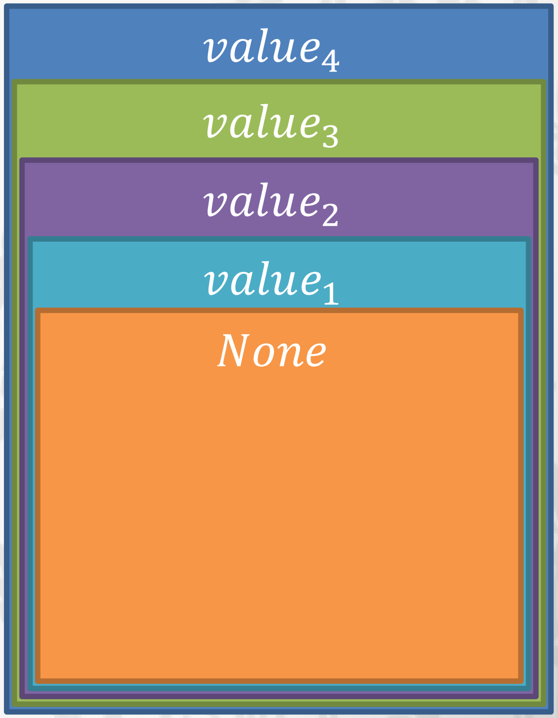
Stack: Operation Push(value)

- Wir können hier eine rekursive Struktur verwenden:
$$stack = (v_n, (v_{n-1}, (v_{n-2}, (\ldots (v_1, None) \ldots))))$$
- Auf das äußerste Element kann direkt zugegriffen werden.
- Alle inneren Elemente erfordern das Auspacken der äußeren Elemente.



$$\xrightarrow{push(value_4)}$$

$$tail = stack$$
$$head = value$$
$$stack = (head, tail)$$

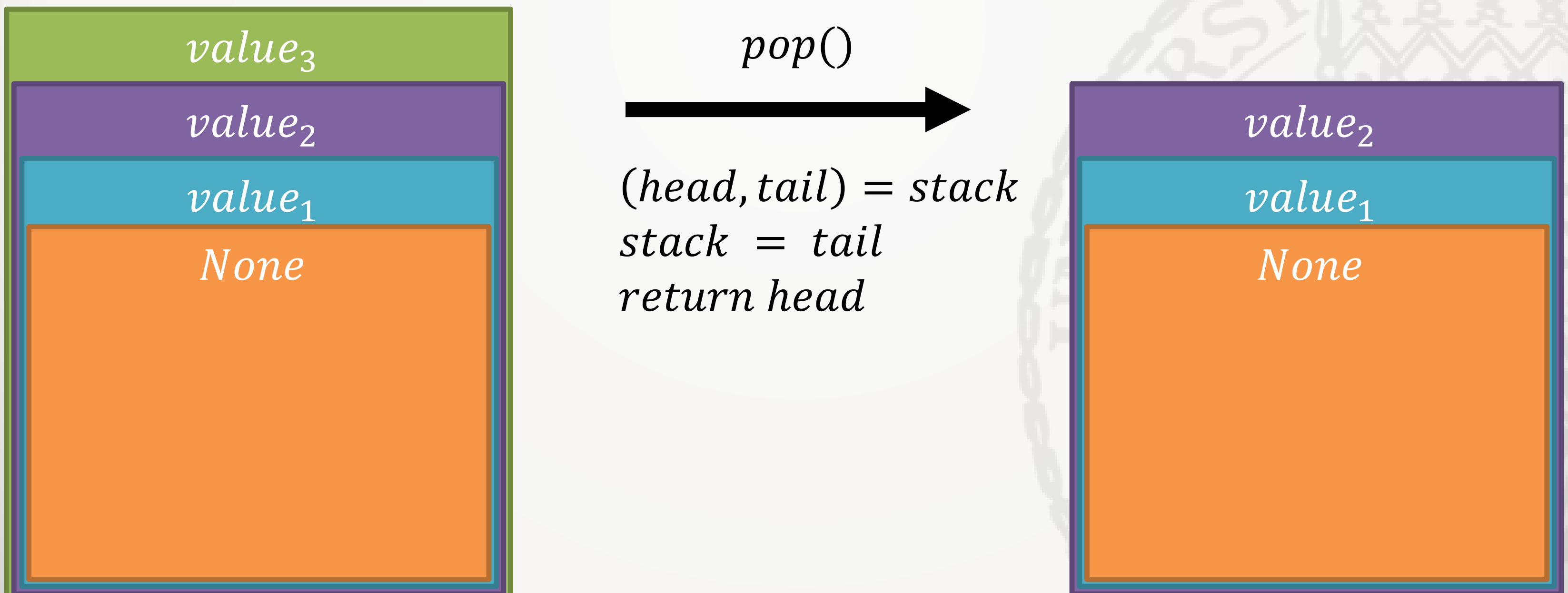


Stack: Operation Pop()

- Wir können hier eine rekursive Struktur verwenden:

$$stack = (v_n, (v_{n-1}, (v_{n-2}, (\ldots (v_1, None) \ldots))))$$

- Auf das äußerste Element kann direkt zugegriffen werden.
- Alle inneren Elemente erfordern das Auspacken der äußeren Elemente.



Stacks in Python

- Wir können hier eine rekursive Struktur verwenden:
$$stack = (v_1, (v_2, (v_3, \dots)))$$
- Auf das äußerste Element kann direkt zugegriffen werden.
- Alle inneren Elemente erfordern das Auspacken der äußeren Elemente.

```
class Stack:
  def __init__(self):
    self.data = None

  def push(self, value):
    self.data = (value, self.data)

  def pop(self):
    if self.data != None:
      head, self.data = self.data
      return head
```

```
# Example
s = Stack()

s.push(3)
s.push(7)
s.push(18)

s.pop() # Out: 18
s.pop() # Out: 7
```

Stacks in Java mit Array

```
Class StackArray implements Stack {
  int top;
  Object[] stack;

  StackArray (int capacity) {
    top = 0;
    stack = new Object[capacity];
  }

  void push (Object v) {
    if(top >= stack.length) {
      // Fehlerbehandlung für
      // Überlauf einfügen
      return;
    } else {
      stack[top] = v;
      top = top + 1;
    }
  }

  Object pop () {
    if (top == 0) {
      // Fehlerbehandlung für
      // Unterlauf einfügen
      return null;
    } else {
      top = top - 1;
      return stack[top];
    }
  }

  boolean isEmpty () {
    return (top == 0);
  }

  boolean isFull () {
    return (top >= stack.length);
  }

} // class StackArray
```

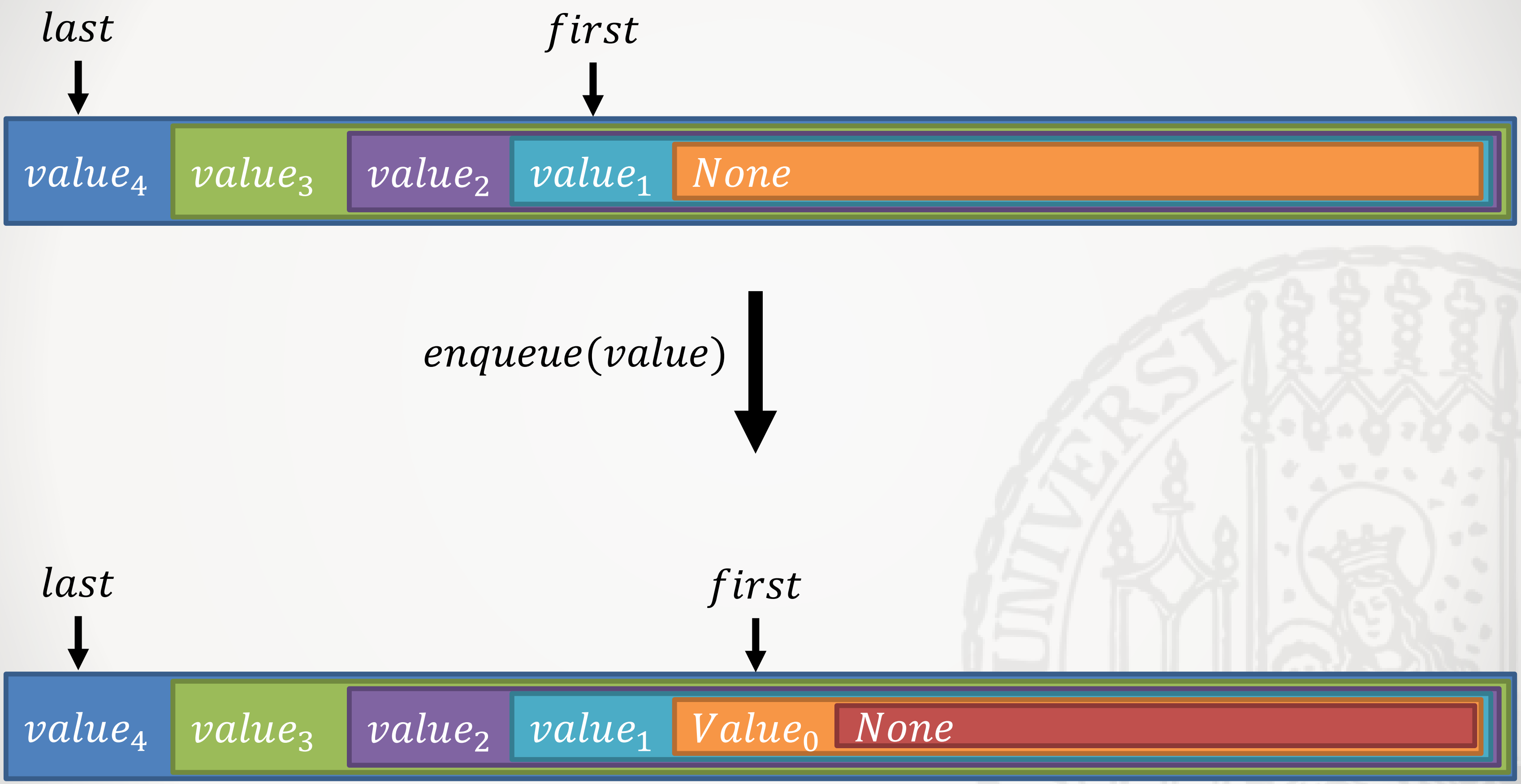
Stacks in Java mit Listen und Pointern

```
Class StackList implements Stack {
  List top;

  StackList () {
    top = null;
  }

  void push (Object v) {
    List elem = new List();
    elem.value = v;
    elem.next = top;
    top = elem;
  }

  Object pop () {
    if (top == null) {
      // Fehlerbehandlung
      // Unterlauf einfügen
      return null;
    } else {
      Object x = top.value;
      top = top.next;
      return x;
    }
  }

  boolean isEmpty () {
    return (top == null);
  }

} // class StackList
```

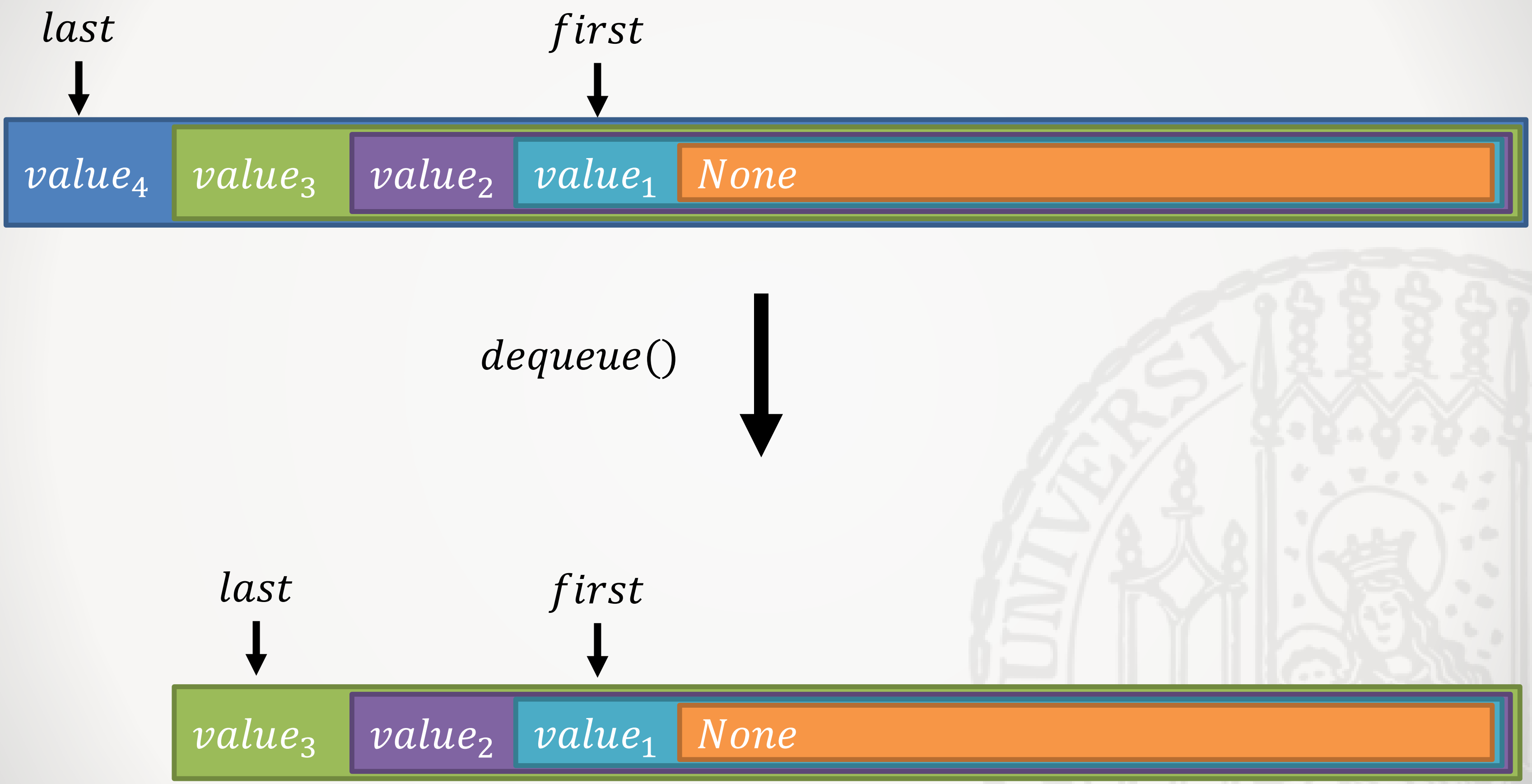
Queue

- Spezifikation
  - Wie Liste: sequentielle Ordnung, aber:
  - Einfügen: Nur am Ende anhängen erlaubt („enqueue“)
  - Auslesen: vorderstes Element zurückgeben („dequeue“)
  - FIFO-Prinzip („First-In-First-Out“)

Queue: Operation enqueue(value)



Queue: Operation dequeue()



Queues in Python

```
class _Entry:
  def __init__(self, value = None, next = None):
    self.value = value
    self.next = next


class Queue:
  def __init__(self):
    self._first = None
    self._last = None


  def enqueue(self, value):
    if self.first == None:
      self.first = self.last = _Entry(value)
    else:
      entry = self.first
      self.first = entry.next = _Entry(value)


  def dequeue(self):
    if self.last != None:
      entry = self.last
      self.last = entry.next
      return entry.value
```

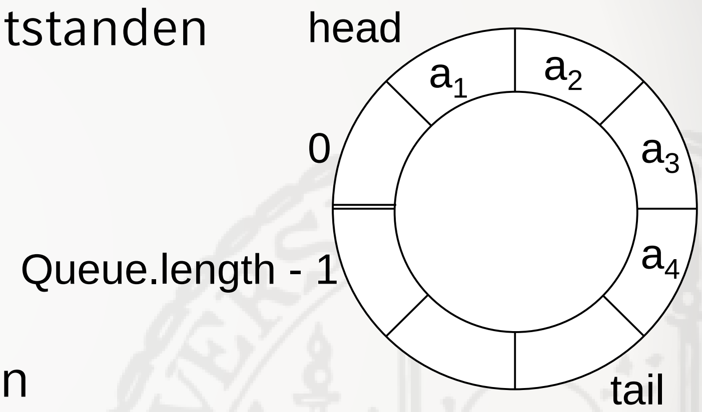
Queues in Java mit Listen

```
class QueueList implements Queue {
 List first, last;

 QueueList () {
  first = new List(); // dummy
  last = first;
 }

 void add (Object v) {
  last.next = new List();
  last.next.value = v;
  last = last.next;
}
```

```
 Object remove () {
  if (first == last) {
   // Fehlerbehandlung Unterlauf
   return null;
  } else {
   Object x = first.value;
   first = first.next;
   return x;
  }
 }

 boolean isEmpty () {
  return (first == last);
 }

} // class QueueList
```

Queue als zyklisches Array

- Versuch: Queue als klassisches (lineares) Array
  - Belegter Bereich „wandert“ von vorne nach hinten durch.
  - Was tun, wenn „enqueue“ hinten anstößt?
  - Durch „dequeue“ ist vorne Platz entstanden
  - Aber: Verschieben ist zu teuer



- Lösungsansatz: „zyklisches“ Array
  - Verbinde Ende mit dem Anfang
  - Ringschluss durch Modulo-Funktion
- Eigenschaften
  - kein Speicher für Pointer nötig
  - leere Elemente (Speicherplatzverlust)
  - Beschränkte Länge

Queues in Java mit (zyklischem) Array

```
class QueueArray implements Queue {
 int first, last;
 Object[] queue;

 QueueArray (int capacity) {
  first = 0;
  last = 0;
  queue = new Object[capacity+1];
 }

 void add (Object v) {
  int next = (last+1) % queue.length;
  if (next == first) {
   // Fehlerbehandlung Überlauf
   return null;
  } else {
   queue[last] = v;
   last = next;
  }
 }
```

```
 Object remove () {
  if (first == last) {
   // Fehlerbehandlung Unterlauf
   return null;
  } else {
   Object x = queue.first;
   first = (first+1) % queue.length;
   return x;
  }
 }

 boolean isEmpty () {
    return (first == last);
  }

  boolean isFull () {
    return (first == (last+1) %
                     queue.length);
  }
} // class QueueArray
```

Priority Queue (Prioritätswarteschlange)

- Charakterisierung
  - Verwalte Elemente mit Prioritätswerten.
  - Verwendung: Algorithmen mit „Bestensuche“
  - Beispiel: Suche nach kürzesten Pfaden in einem Netzwerk
  - Statt FIFO (push-pop) und LIFO (enqueue-dequeue)

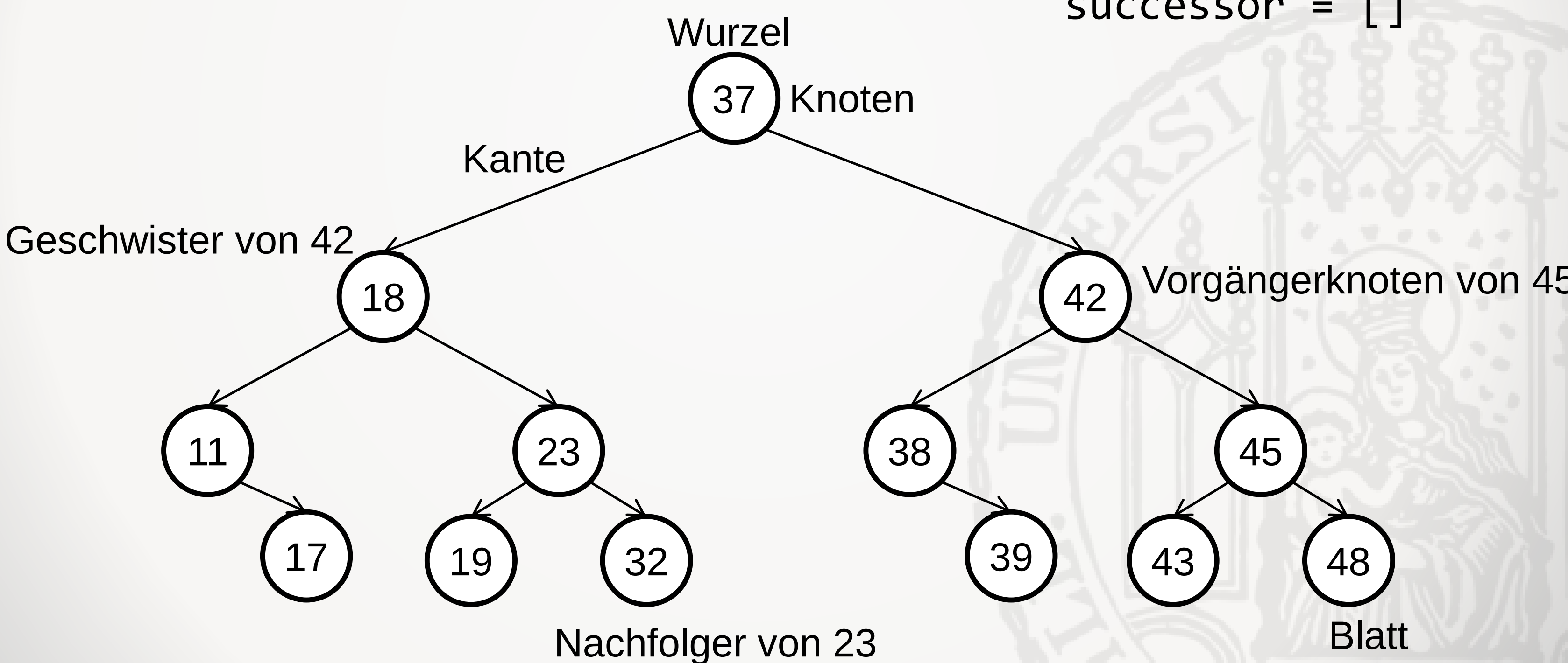
- Operationen
  - Insert (elem, prio) --- füge Element mit Priorität ein.
  - GetMin () --- Liefere Element mit der höchsten Priorität und entferne es aus der Menge.

- Bemerkungen
  - Priorität kann Minimum (Kosten) oder Maximum (Score) sein.
  - Implementierung durch Heapstrukturen üblich (siehe später).

Bäume

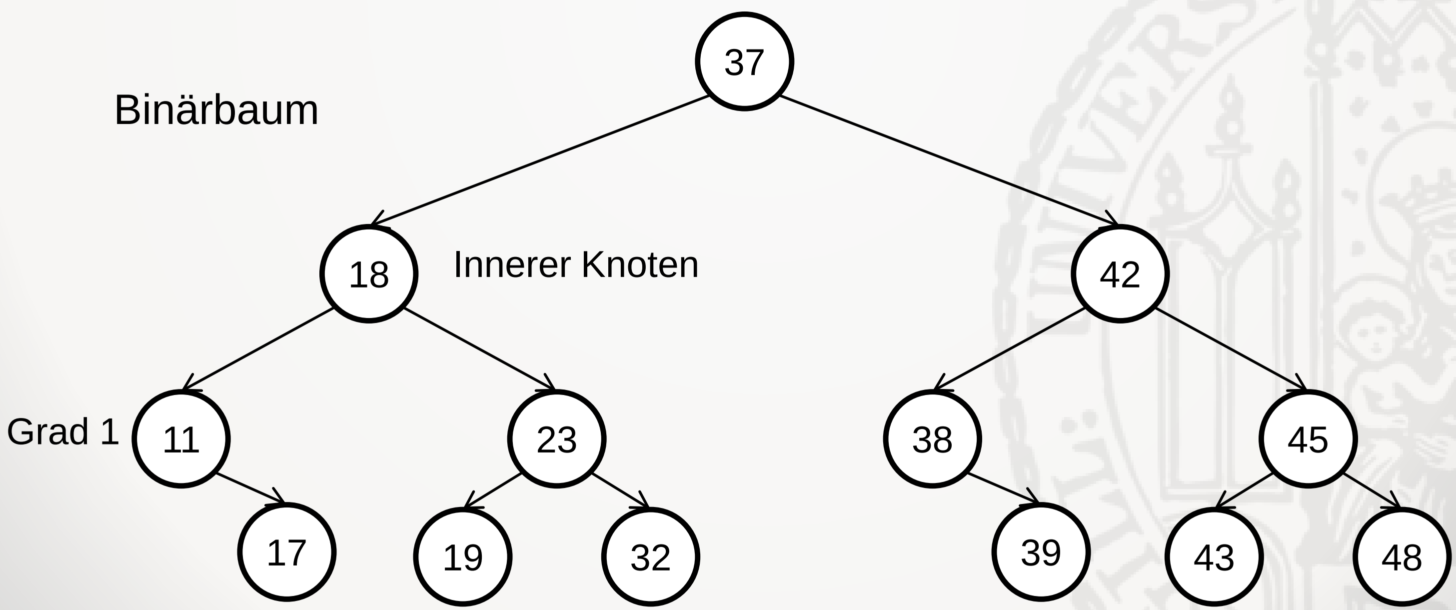
- Erweitern wir das Listenkonzept und erlauben mehrere Nachfolger, sprechen wir von Bäumen
- Relationen in Bäumen definieren eine Hierarchie

```
class Node:
  def __init__(self):
    data = None
    successor = []
```



Terminologie für Bäume

- Pfad: Folge von Knoten, die durch Kanten direkt verbunden sind
- Pfadlänge: Anzahl der Kanten eines Pfades („Kantenlänge“)
- Grad eines Knotens: Anzahl der unmittelbaren Nachfolger
- Verzweigungsgrad eines Baums: Maximum aller Knotengrade
  - Ein Baum mit Verzweigungsgrad zwei heißt „Binärbaum“



Eigenschaften von Bäumen

- Kantenmaximalität: Ein Baum mit n Knoten hat genau $n-1$ Kanten.
  - Entfernt man eine Kante, so ist der Baum nicht mehr zusammenhängend.
  - Fügt man eine Kante hinzu, ist der Baum nicht mehr zyklenfrei.
- Vollständiger Baum: Jeder Knoten hat maximalen Grad.
- Die Höhe eines Baums ist die Länge des längsten Pfads von der Wurzel zu einem Blatt.

  ACHTUNG: Die Definition der Höhe ist in der Literatur nicht einheitlich gewählt, man unterscheidet:
  - Höhe als Anzahl **Kanten** auf längstem Pfad von Wurzel zu Blatt.
  - Höhe als Anzahl **Knoten** auf längstem Pfad von Wurzel zu Blatt.

Beziehung zwischen Höhe und Knoten

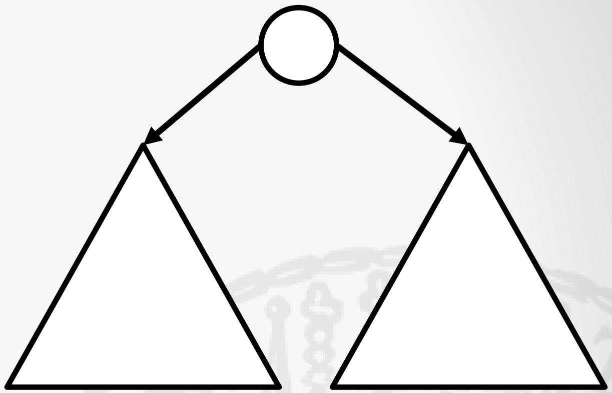
Für einen Binärbaum t der Höhe h gilt:

i. t hat maximal $2^{h+1} - 1$ Knoten.

ii. t hat mindestens $h + 1$ Knoten

iii. t hat maximal $2^h - 1$ innere Knoten.

iv. t hat maximal 2^h Blätter.



Beweis (i) per vollständiger Induktion:

- *Induktionsbeginn:* Ein Baum der Höhe $h = 0$ besteht nur aus der Wurzel und hat $2^{0+1} - 1 = 1$ Knoten.
- *Induktionsannahme:* Ein Binärbaum der Höhe h hat maximal $2^{h+1} - 1$ Knoten.
- *Induktionsschritt:* Ein Binärbaum der Höhe $h' = h + 1$ hat eine Wurzel und zwei Teilbäume der Höhe h (einer dürfte kleiner sein), und damit insgesamt bis zu $1 + 2 \cdot (2^{h+1} - 1) = 2^{h+2} - 1$ Knoten.

Beziehung zwischen Höhe und Knoten

Für einen Binärbaum t der Höhe h gilt:

i. t hat maximal $2^{h+1} - 1$ Knoten.

ii. t hat mindestens $h + 1$ Knoten

iii. t hat maximal $2^h - 1$ innere Knoten.

iv. t hat maximal 2^h Blätter.

Beweis (ii): trivial
Ein entarteter Baum der Höhe h, der aus einer (linearen) Folge von Knoten mit genau einem Nachfolger besteht, hat $h + 1$ viele Knoten.

Beziehung zwischen Höhe und Knoten

Für einen Binärbaum t der Höhe h gilt:

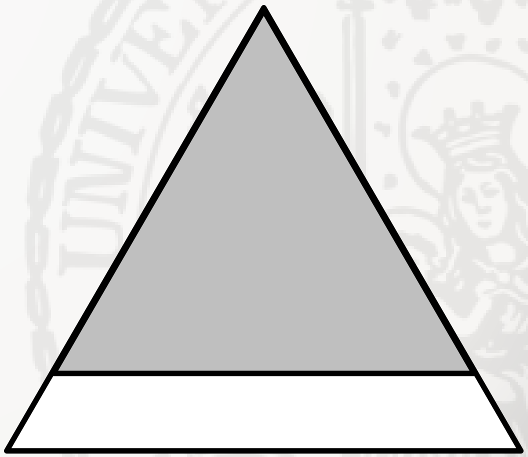
i. t hat maximal $2^{h+1} - 1$ Knoten.

ii. t hat mindestens $h + 1$ Knoten

iii. t hat maximal $2^{h} - 1$ innere Knoten.

iv. t hat maximal 2^{h} Blätter.

Beweis: (iii)
Die Menge der inneren Knoten entspricht dem verbleibendem Baum, wenn man alle Blätter abschneidet. Der abgeschnittene Baum hat die Höhe $h - 1$ und gemäß (i) $2^{(h-1)+1} - 1 = 2^{h} - 1$ viele Knoten.



Beziehung zwischen Höhe und Knoten

Für einen Binärbaum t der Höhe h gilt:

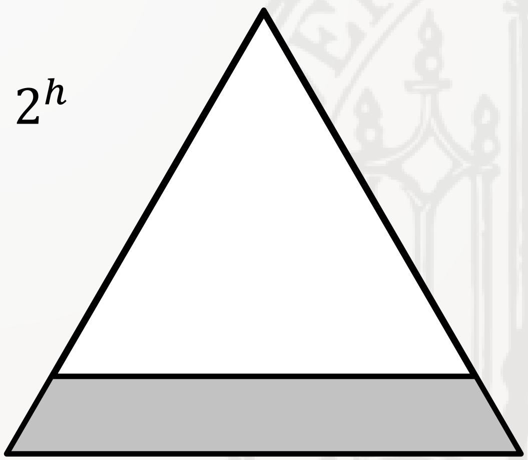
i. t hat maximal $2^{h+1} - 1$ Knoten.

ii. t hat mindestens $h + 1$ Knoten

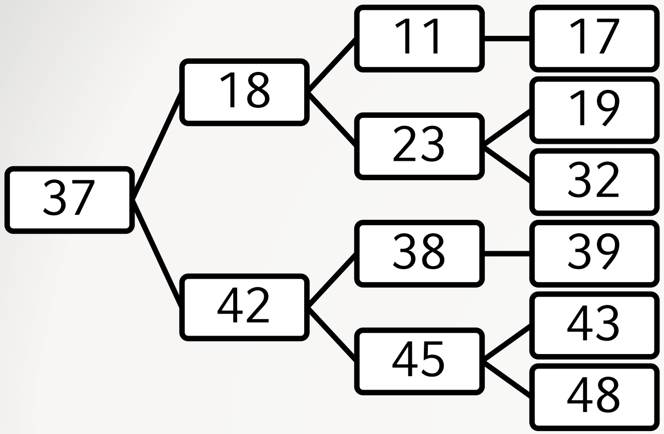
iii. t hat maximal $2^h - 1$ innere Knoten.

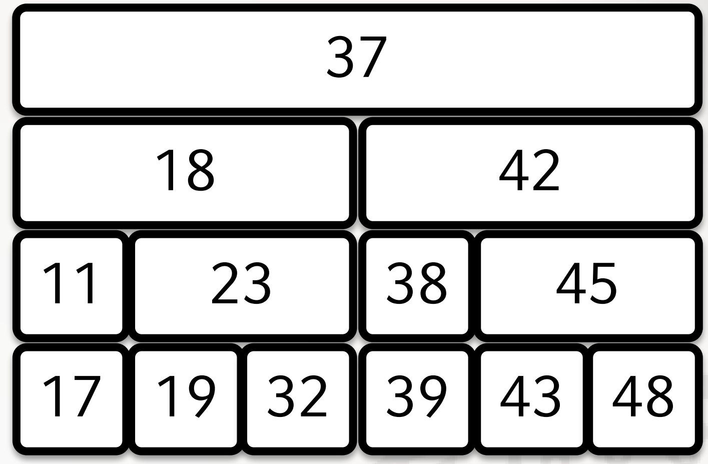
iv. t hat maximal 2^h Blätter.

Beweis: (iv)
Folgt aus (i) und (iii):
$\left(2^{h+1} - 1\right) - \left(2^h - 1\right) = 2 \cdot 2^h - 2^h = 2^h$

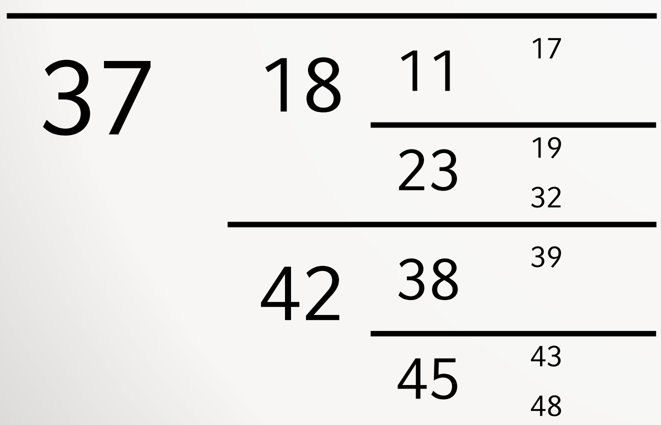


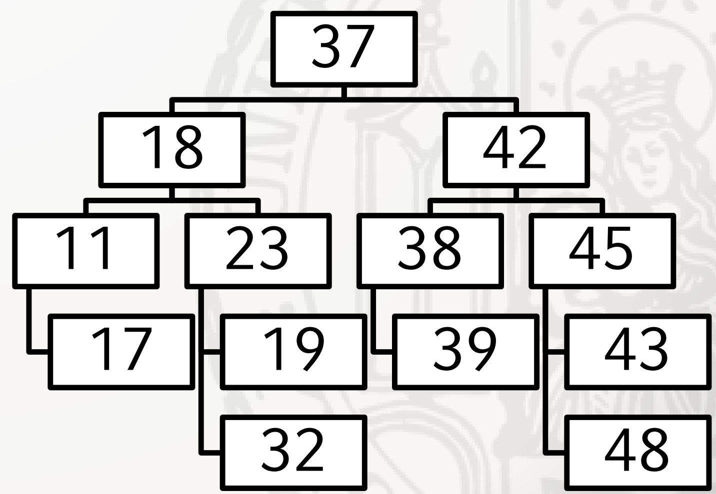
Alternative Baumdarstellungen





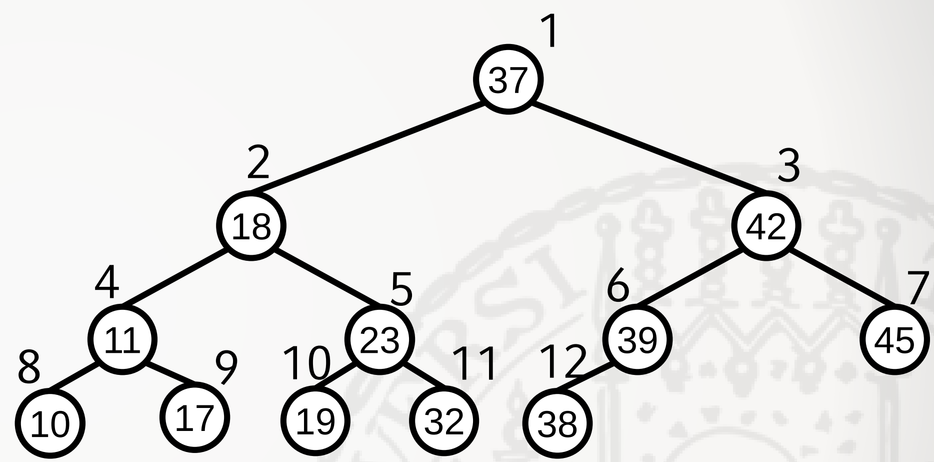
$$37\left(18\left(11(17), 23(19,32)\right), 42\left(38(39), 45(43,48)\right)\right)$$





Arrayeinbettung

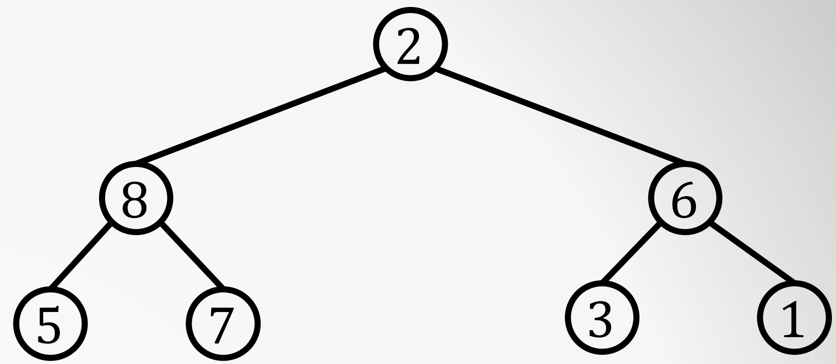
- Wir wissen: Ein Binärbaum der Höhe h hat $n \leq 2^{h+1} - 1$ Knoten.
- Ein Array der Größe n kann daher einen Binärbaum speichern
  - Ebenen von der Wurzel an in das Array eintragen
  - Leere Positionen im Array freilassen
- Kinder von Knoten i:
  - $2i$, $2i + 1$
- Vorgänger von Knoten i:
  - $\lfloor i/2 \rfloor$



1	2	3	4	5	6	7	8	9	10	11	12
37	18	42	11	23	39	45	10	17	19	32	38

- Knoten $\frac{n}{2} < i \leq n$ sind Blätter, da Knoten $2i > n$ nicht existieren.
- Hier: Array beginnt bei 1 (Platz 0 verschenken lohnt sich).

Baumtraversierungen



- Tiefendurchlauf (depth first):
  - Durchlaufe zu jedem Knoten rekursiv die Teilbäume von links nach rechts
  - Preorder/Präfix: notiere erst einen Knoten, dann seine Teilbäume
    - Beispiel: 2 8 5 7 6 3 1 [polnische Notation bei Termen]
  - Postorder/Postfix: notiere erst Teilbäume eines Knotens, dann ihn selbst
    - Beispiel: 5 7 8 3 1 6 2
  - Inorder/Infix: notiere 1. Teilbaum, dann Knoten selbst, dann restliche Teilbäume
    - Beispiel: 5 8 7 2 3 6 1 [Mehrdeutigkeit möglich]
- Breitendurchlauf (breadth first):
  - Knoten ebenenweise durchlaufen, von links nach rechts
    - Beispiel: 2 8 6 5 7 3 1

- Alle Durchläufe auf beliebigen Bäumen durchführbar
  - Inorder-Notation nur auf Binärbäumen gebräuchlich

Zusammenfassung Grundlagen

- Probleme und Instanzen
- Algorithmen
  - Definition
  - Darstellungen (Prosa, Pseudocode, Programmcode)
  - Eigenschaften
- Grundlegende Datenstrukturen
  - Arrays
  - Listen
    - Stacks
    - Queues
  - Bäume
    - Eigenschaften
    - Binärbäume
    - Traversierungen
    - Heaps