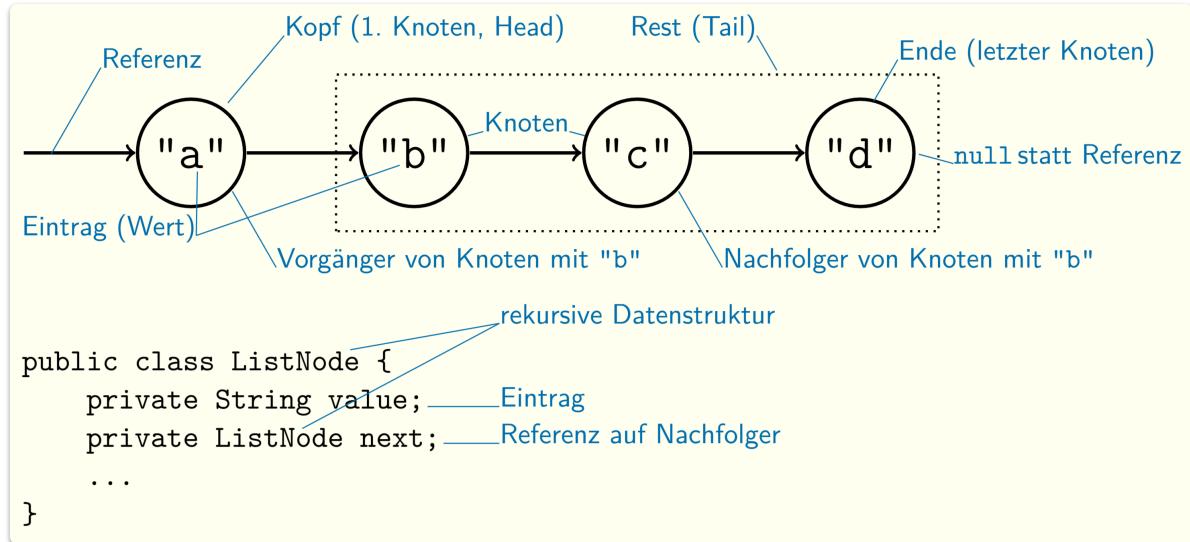


4. Lineare Liste und co

Quelle: ep2-04_lineare-Liste_Binäre-Bäume.pdf

Beinhaltet: Lineare Liste, Linearer Baum

Lineare Liste

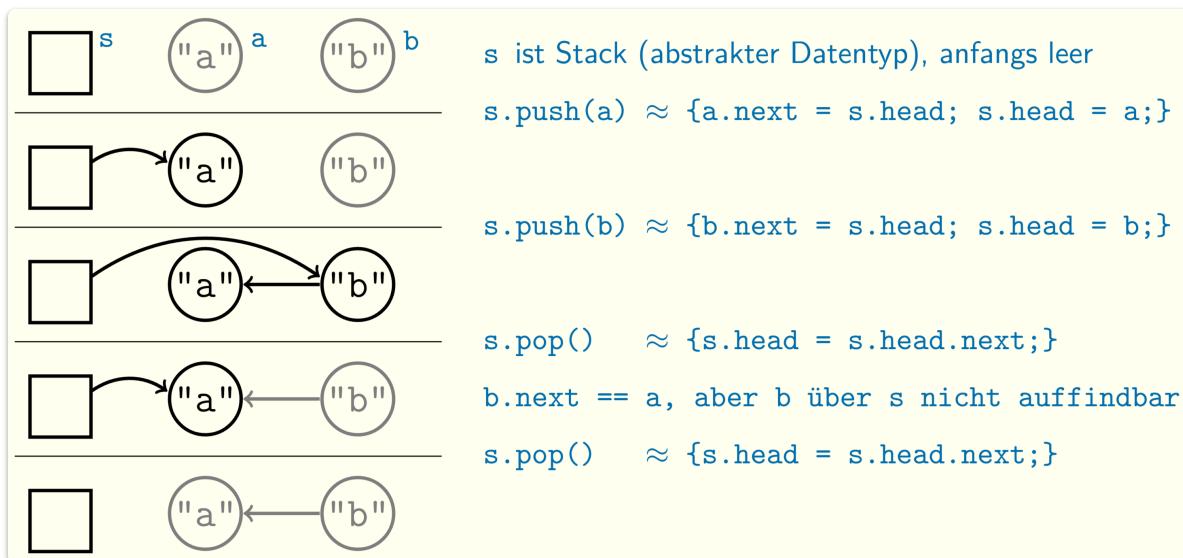


das ist aber falsch weil das mit Tail ist nur der letzte gemeint und nicht der ganze Kasten

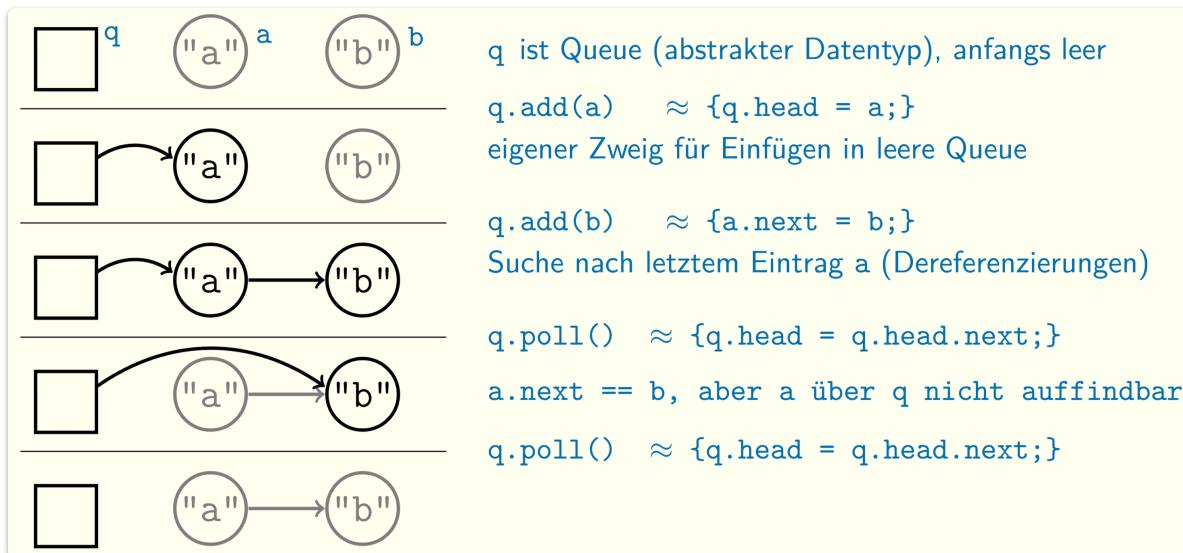
ist ein 1. Einführung > Abstrakte Datentypen

Arten von Listen

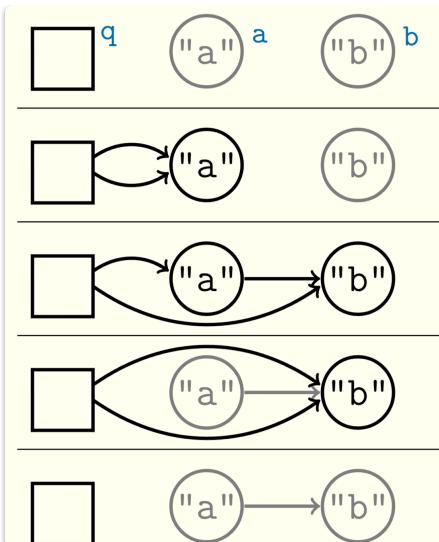
Lineare Liste als Stack



Lineare Liste als Queue



Lineare Liste als Queue (mit last)



`q` ist Queue (abstrakter Datentyp), anfangs leer

`q.add(a)` $\approx \{q.\text{head} = a; q.\text{last} = a;\}$

eigener Zweig für Einfügen in leere Queue

`q.add(b)` $\approx \{q.\text{last} = q.\text{last}.\text{next} = b;\}$

keine Suche nach letztem Eintrag

`q.poll()` $\approx \{q.\text{head} = q.\text{head}.\text{next};\}$

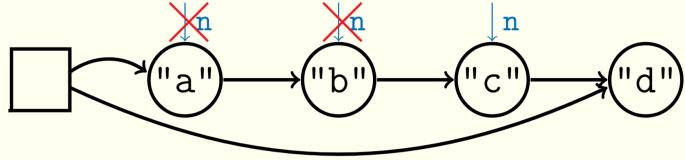
`a.next == b`, aber `a` über `q` nicht auffindbar

`q.poll()` $\approx \{q.\text{head} = q.\text{last} = \text{null};\}$

eigener Zweig für Entfernen des letzten Knotens

Methoden auf Listen

Traversieren einer Liste (Suche)



```

ListNode n = head;           n könnte bereits zu Beginn gleich null sein
while (n != null && ...) {   Suchbedingung
    ...
    n = n.next();           was für jeden besuchten Knoten gemacht werden soll
}                           dereferenzieren (weiterschalten)

```

Einfügen in eine Liste



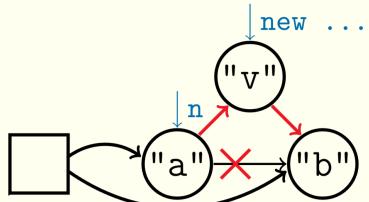
Einfügen in eine Liste

```

if (... /* insert at begin */) {
    head = new ListNode(v, head);
    if (last == null) {
        last = head;
    }
} else { /* not at begin */
    ListNode n = ...; /* insert after n */
    n.setNext(new ListNode(v, n.next()));
    if (last == n) {
        last = n.next();
    }
}

```

new ...
 Sonderbehandlung für ersten Eintrag
 Referenz auf Vorgänger
 Zusatzaufwand für last



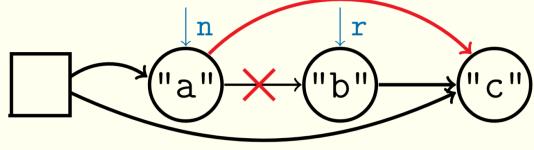
Entfernen aus einer Liste

nur wenn zu entfernender Knoten existiert

```

if (... /* remove first node */) {
    head = head.next();
    if (head == null) {
        last = null;
    }
} else {
    ListNode n = ...; /* predecessor of r */
    ListNode r = n.next(); /* to be removed */
    n.setNext(r.next());
    if (last == r) {
        last = n;
    }
}

```



Sonderbehandlung für ersten Eintrag

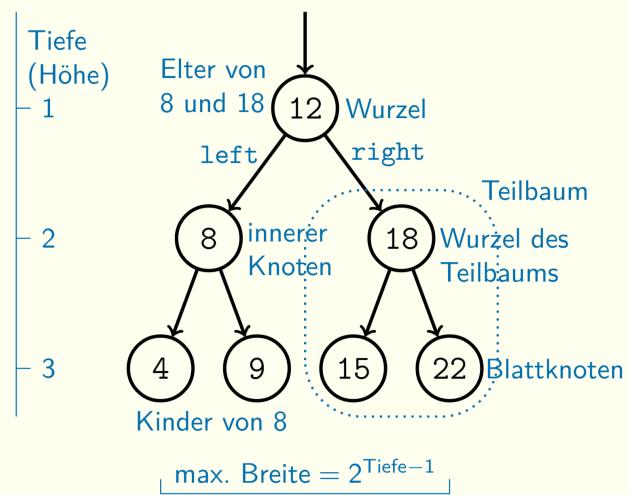
Referenz auf Vorgänger

Zusatzaufwand für last

Linearer Baum

jeder Knoten hat bis zu 2 Kinder

```
public class TreeNode {
    private int value;
    private TreeNode left;
    private TreeNode right;
    ...
}
```



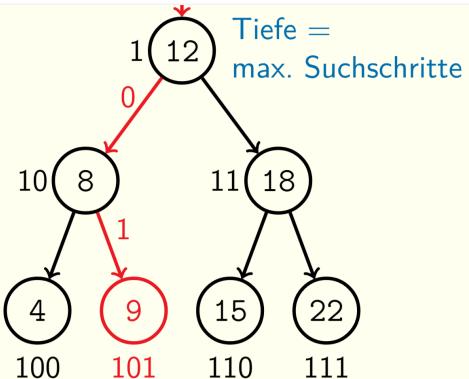
ist ein 1. Einführung > Abstrakte Datentypen

Binärer Suchbaum

Binärer Baum, Einträge sortiert
(z.B. links kleiner, rechts größer/gleich)

Analogie zu Binärzahlen – effiziente Suche

```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public boolean contains(int v) {
        return v == value ||
               (v < value ? left!=null && left.contains(v)
jeweils nur ein Zweig betrachtet : right!=null && right.contains(v));
    }
}
```

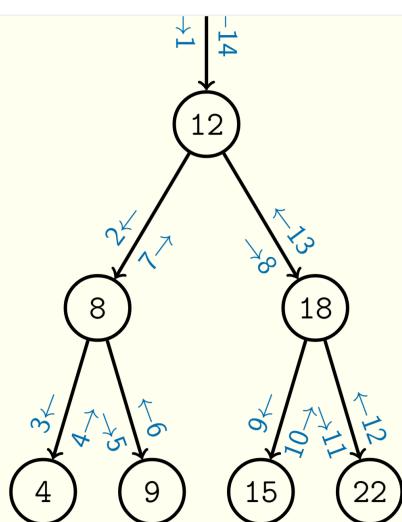


Einfügen in binären Suchbaum

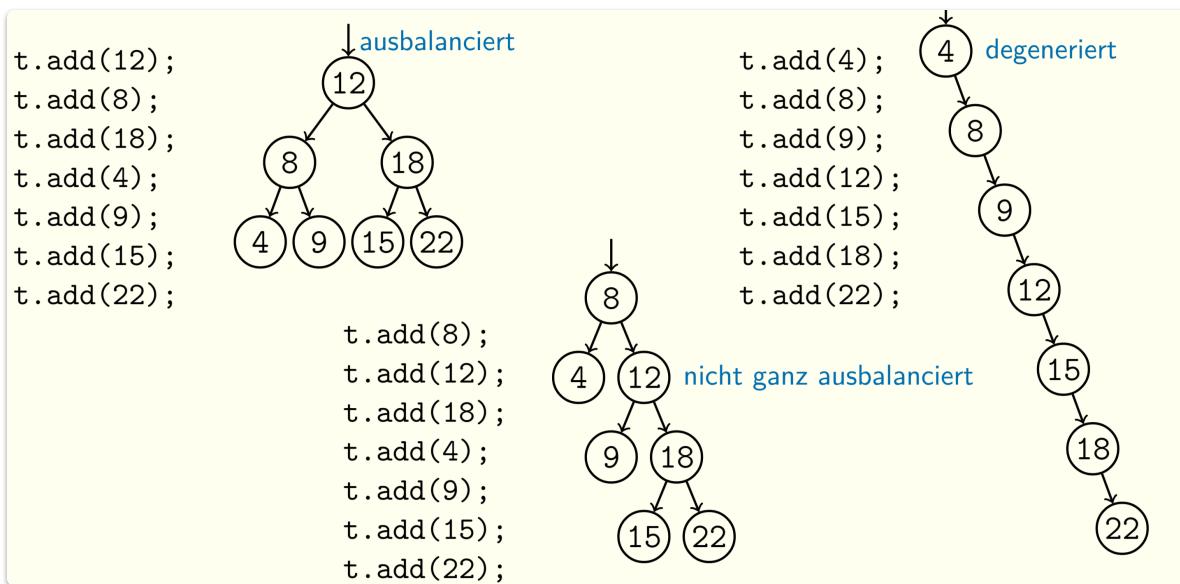
```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public void add(int v) {
        if (v < value) { // links oder rechts?
            if (left != null) { // wenn Teilbaum existiert, rekursiver Aufruf
                left.add(v); // sonst Platz zum Einfügen gefunden
            } else { left = new TreeNode(v); }
        } else if (v > value) { // nicht einfügen wenn schon vorhanden,
            if (right != null) { right.add(v); }
            else { right = new TreeNode(v); }
        }
    }
}
```

Traversieren des gesamten Baums

```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public void visit() {
        ... in Reihenfolge ↓ (12,8,4,9,18,15,22)
        if (left != null) { left.visit(); }
        ... in sortierter Reihenfolge (4,8,9,12,15,18,22)
        ... Vertauschen kehrt Sortierreihenfolge um
        if (right != null) { right.visit(); }
        ... in Reihenfolge ↑ (4,9,8,15,22,18,12)
    }
}
```



Baumstruktur von Reihenfolge des Einfügens abhängig



Binärer Suchbaum benötigt Sortierbarkeit

```
public class TANode {
    private String key, value;
    private TANode left, right;
    ...
    private int compare(String k) {   Vergleich nur mit key, nicht mit value
        if (k == null) {
            return key == null ? 0 : -1;
        }
        if (key == null) {           x.compareTo(y) → -1 wenn x kleiner y
            return 1;                0 wenn x gleich y
        }
        return k.compareTo(key);
    }
}
```

1 wenn x größer y

Rekursion

Rekursion: Grundprinzipien

- **Fundierung:**
 - Rekursion muss immer eine **Abbruchbedingung** haben, um die unendliche Rekursion zu verhindern.
 - **Abbruchbedingung:** Ein Zustand, in dem die Rekursion nicht weiter fortgesetzt wird.
Beispiel: Ein Basisfall in einer rekursiven Methode (z. B. bei der Berechnung einer Fakultät, wenn der Wert 0 oder 1 erreicht wird).
- **Fortschritt:**
 - In jedem rekursiven Aufruf müssen **andere Parameterwerte** verwendet werden, die dem Problem näher kommen.
 - Dies bedeutet, dass sich der Zustand in jedem Schritt der Rekursion verändern muss, damit der Basisfall irgendwann erreicht wird.

Beispiel einer rekursiven Methode:

```
public int factorial(int n) {
    if (n <= 1) {
        return 1; // Fundierung (Abbruchbedingung)
    }
    return n * factorial(n - 1); // Fortschritt durch andere Parameterwerte
}
```



- **Fundierung:** Wenn `n <= 1` erreicht wird, hört die Rekursion auf.
- **Fortschritt:** Der Parameter `n` wird in jedem rekursiven Aufruf um 1 verringert, bis die Abbruchbedingung erreicht wird.

Rekursive Datenstrukturen

Rekursive Datenstrukturen: Fundierung und Fortschritt

- **Fundierung:**
 - Die **Fundierung** einer rekursiven Datenstruktur erfolgt durch **null** oder durch einen **spezifischen Knoten**, der als Basisfall dient.
 - In vielen Fällen ist `null` der Ausgangspunkt, der das Ende einer rekursiven Struktur markiert (z. B. das Ende einer verknüpften Liste).
- **Fortschritt:**
 - **Fortschritt** erfolgt durch **induktiven Aufbau**, das heißt durch **schrittweises Hinzufügen von Knoten** und **Referenzen**.

- Bei der Rekursion werden Elemente schrittweise hinzugefügt, bis der Basisfall erreicht wird (z. B. ein `null`-Knoten).

Beispiel einer rekursiven Datenstruktur (z. B. verkettete Liste):

```
public class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null; // Fundierung: der letzte Knoten verweist auf null
    }
}

public void printList(Node head) {
    if (head == null) {
        return; // Fundierung (Abbruchbedingung)
    }
    System.out.println(head.data);
    printList(head.next); // Fortschritt (rekursiver Aufruf auf das nächste
Element)
}
```



- Fundierung:** Der `null`-Wert am Ende der Liste stellt den Abbruch der Rekursion dar (kein weiteres Element mehr).
- Fortschritt:** Der rekursive Aufruf geht von einem Knoten zum nächsten, wobei `head.next` auf den nächsten Knoten verweist, bis der Endknoten erreicht wird.

Rekursive Methoden und Datenstrukturen gekoppelt

- Verknüpfung von rekursiven Methoden und rekursiven Datenstrukturen:**
 - Oft sind **rekursive Methoden** und **rekursive Datenstrukturen** miteinander gekoppelt.
 - Die Abbruchbedingung in der Methode wird durch das Erreichen von `null` oder eines speziellen Knotens in der Datenstruktur ausgelöst.
 - Der **Fortschritt** erfolgt durch **Dereferenzierung** der Knoten bei rekursiven Aufrufen (z. B. das Weitergeben der Referenz zum nächsten Knoten in einer Liste).

Beispiel für verknüpfte rekursive Methode und Datenstruktur:

- Bei der rekursiven Traversierung einer Liste wird der Fortschritt durch Dereferenzieren der `next`-Referenz im aktuellen Knoten erreicht.

