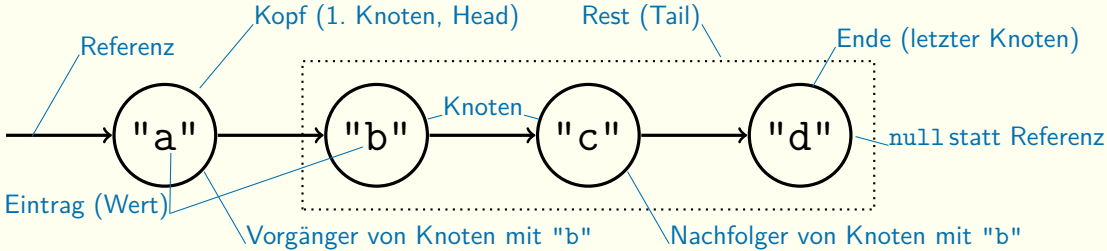
Lineare Liste
Binärer Baum

Lineare Liste



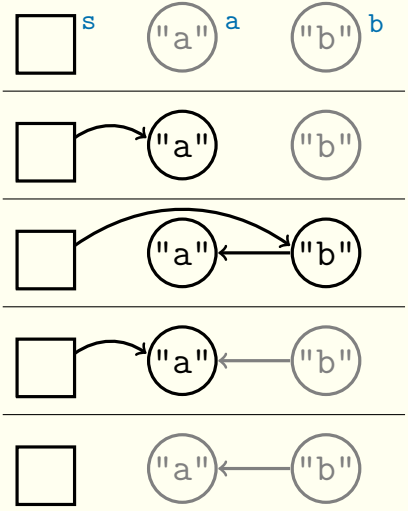
rekursive Datenstruktur

```
public class ListNode {
    private String value;
    private ListNode next;
    ...
}
```

Eintrag

Referenz auf Nachfolger

Lineare Liste als Stack



s ist Stack (abstrakter Datentyp), anfangs leer

`s.push(a)` ≈ `{a.next = s.head; s.head = a;}`

`s.push(b)` ≈ `{b.next = s.head; s.head = b;}`

`s.pop()` ≈ `{s.head = s.head.next;}`

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

`s.pop()` ≈ `{s.head = s.head.next;}`

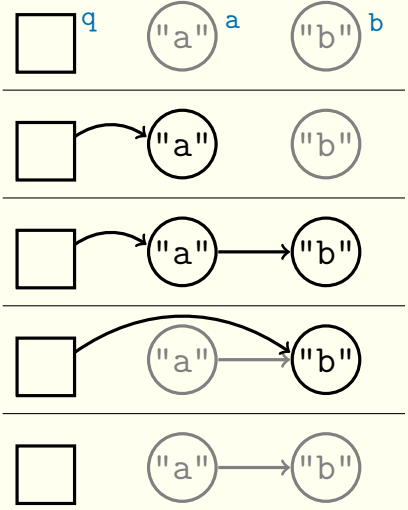
Aufgabe: Eintrag in mehreren Listen

In Gruppen zu 2 bis 3 Personen:

Kann ein Eintrag gleichzeitig in mehreren unterschiedlichen Listen enhalten sein?
Wie bzw. warum ist das (nicht) möglich?

Zeit: 2 Minuten

Lineare Liste als Queue (mit linearer Suche)



q ist Queue (abstrakter Datentyp), anfangs leer

`q.add(a)` ≈ `{q.head = a;}`

eigener Zweig für Einfügen in leere Queue

`q.add(b)` ≈ `{a.next = b;}`

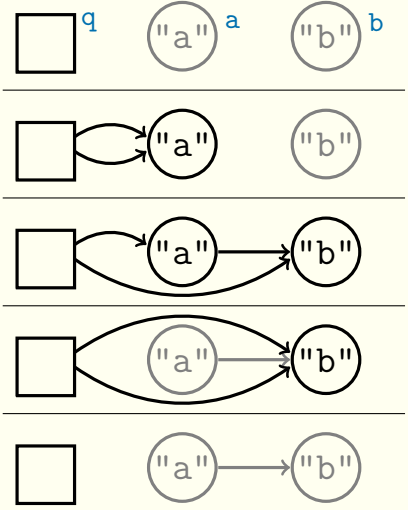
Suche nach letztem Eintrag a (Dereferenzierungen)

`q.poll()` ≈ `{q.head = q.head.next;}`

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

`q.poll()` ≈ `{q.head = q.head.next;}`

Lineare Liste als Queue (mit `last`)



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

`q.add(a)` ≈ `{q.head = a; q.last = a;}`
eigener Zweig für Einfügen in leere Queue

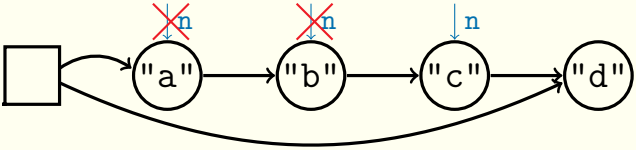
`q.add(b)` ≈ `{q.last = q.last.next = b;}`
keine Suche nach letztem Eintrag

`q.poll()` ≈ `{q.head = q.head.next;}`

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

`q.poll()` ≈ `{q.head = q.last = null;}`
eigener Zweig für Entfernen des letzten Knotens

Traversieren einer Liste (Suche)




```
ListNode n = head;
while (n != null && ...) {
    ...
    n = n.next();
}
```

n könnte bereits zu Beginn gleich `null` sein

Suchbedingung

was für jeden besuchten Knoten gemacht werden soll

dereferenzieren (weiterschalten)

Aufgabe: `null`-Abfragen in rekursiven Datenstrukturen

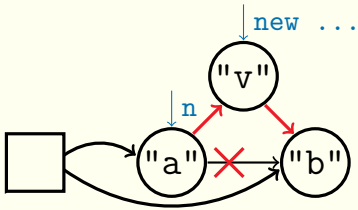
In Gruppen zu 2 bis 3 Personen:

Beim Traversieren von Listen (und anderen rekursiven Datenstrukturen) sind häufig `null`-Abfragen nötig? Warum ist das so, und wo kann `null` vorkommen?

Zeit: 2 Minuten

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();
    }
}
```



Sonderbehandlung für ersten Eintrag

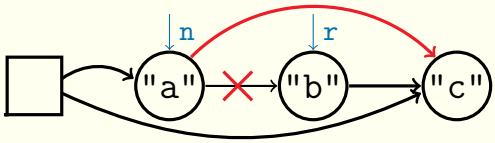
Referenz auf Vorgänger

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`

Aufgabe: Lineare Liste versus Array

In Gruppen zu 2 bis 3 Personen:

Stellen Sie sich vor, Sie hätten kein Array zur Verfügung.

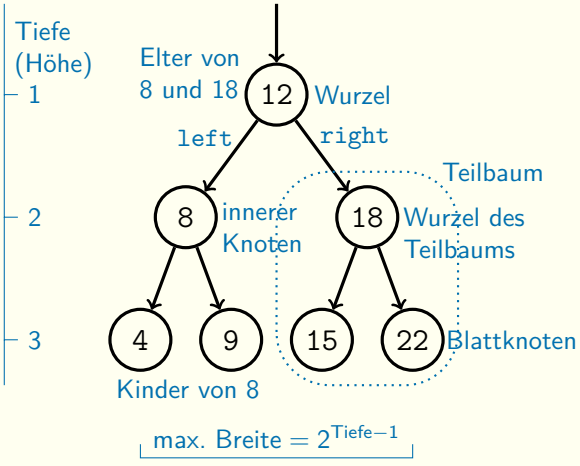
Könnte man die Funktionalität eines Arrays über lineare Listen simulieren?

Zeit: 2 Minuten

Binärer Baum

jeder Knoten hat bis zu 2 Kinder

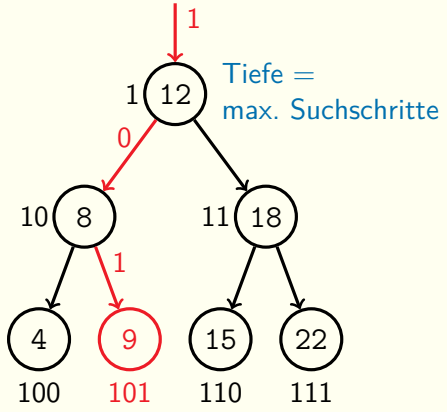
```
public class TreeNode {
    private int value;
    private TreeNode left;
    private TreeNode right;
    ...
}
```



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)
                           : right!=null && right.contains(v));
    }
}
```

jeweils nur ein Zweig betrachtet

Einfügen in binären Suchbaum

```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public void add(int v) {
        if (v < value) {
            if (left != null) { left.add(v); }
            else { left = new TreeNode(v); }
        } else if (v > value) {
            if (right != null) { right.add(v); }
            else { right = new TreeNode(v); }
        }
    }
}
```

links oder rechts?

wenn Teilbaum existiert, rekursiver Aufruf

sonst Platz zum Einfügen gefunden

nicht einfügen wenn schon vorhanden,
Bedingung weglassen für mehrfache Vorkommen

Aufgabe: Wie viel effizienter ist Suche im Baum?

In Gruppen zu 2 bis 3 Personen:

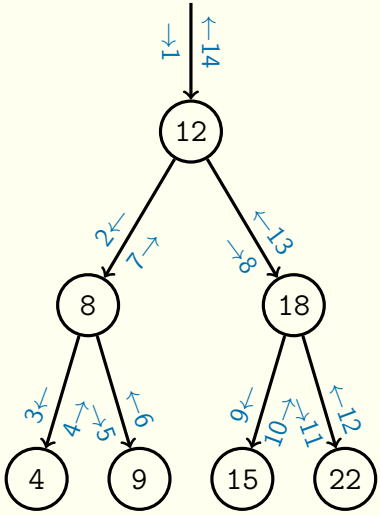
Bei der Suche im (und beim Einfügen in den) binären Suchbaum wird jeweils nur einer der beiden Kinder besucht.

Bedeutet das, dass nur ca. 50% der Knoten im Baum besucht werden, oder sind es mehr bzw. weniger?

Zeit: 2 Minuten

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)
    }
}
```



Rekursion

nur sinnvoll mit **Fundierung** und **Fortschritt**

rekursive **Methoden:** Fundierung durch **Abbruchbedingung**
Fortschritt durch **andere Parameterwerte** in jedem Aufruf

rekursive **Datenstrukturen:** Fundierung durch `null` oder **spezielle Knoten**
Fortschritt durch **induktiven Aufbau**
(schrittweises Hinzufügen von Knoten und Referenzen)

rekursive Methoden oft mit rekursiven Datenstrukturen gekoppelt:

- Abbruchbedingung ist **Erreichen von `null` oder speziellem Knoten**
- Fortschritt durch **Dereferenzieren** bei rekursiven Aufrufen

Aufgabe: Dereferenzierung bei rekursiven Aufrufen

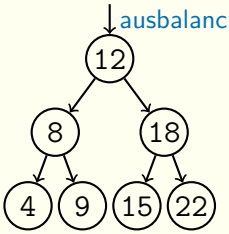
In Gruppen zu 2 bis 3 Personen:

Wo wird in rekursiven Aufrufen von `add` bzw. `contains` im Baum dereferenziert?

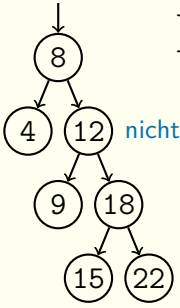
Zeit: 2 Minuten

Baumstruktur von Reihenfolge des Einfügens abhängig

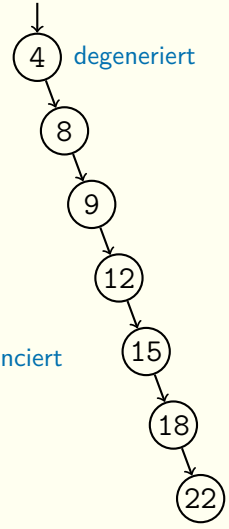
```
t.add(12);
t.add(8);
t.add(18);
t.add(4);
t.add(9);
t.add(15);
t.add(22);
```




```
t.add(8);
t.add(12);
t.add(18);
t.add(4);
t.add(9);
t.add(15);
t.add(22);
```




```
t.add(4);
t.add(8);
t.add(9);
t.add(12);
t.add(15);
t.add(18);
t.add(22);
```



Binärer Suchbaum benötigt Sortierbarkeit

```
public class TANode {
    private String key, value;
    private TANode left, right;
    ...
    private int compare(String k) {
        if (k == null) {
            return key == null ? 0 : -1;
        }
        if (key == null) {
            return 1;
        }
        return k.compareTo(key);
    }
```

Vergleich nur mit `key`, nicht mit `value`

`x.compareTo(y)` → `-1` wenn `x` kleiner `y`
`0` wenn `x` gleich `y`
`1` wenn `x` größer `y`