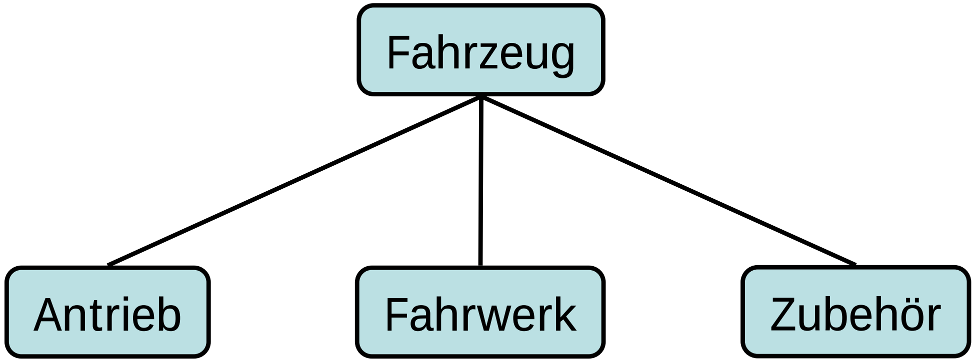
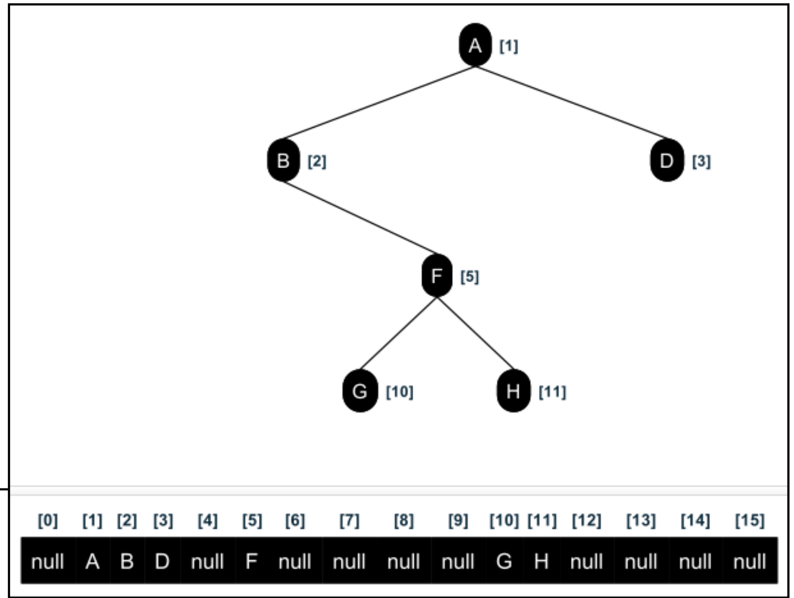
Ziele dieser Woche

- Sie verstehen wie Bäume aufgebaut sind.
- Sie wissen wie Bäume gespeichert werden.
- Sie können folgende Traversierungen anwenden:
    - *Preorder*
    - *Postorder*
    - *Breadth-First*
    - *Inorder*
- Sie können einen Array-basierten Baum implementieren.

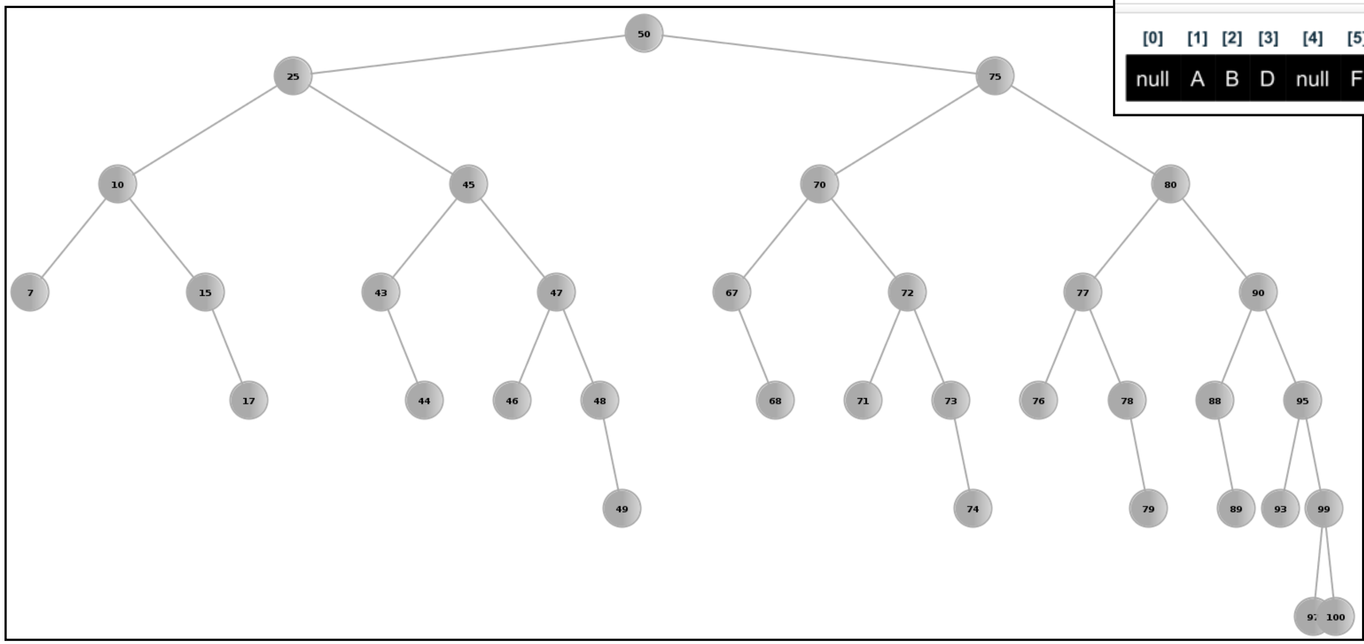
Trees / Bäume



Fahrzeug
Antrieb
Fahrwerk
Zubehör



A [1]
B [2]
D [3]
F [5]
G [10]
H [11]
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]
null A B D null F null null null null G H null null null null



50
25
75
10
45
70
80
7
15
43
47
67
72
77
90
17
44
46
48
68
71
73
76
78
88
95
49
74
79
89
93
99
100

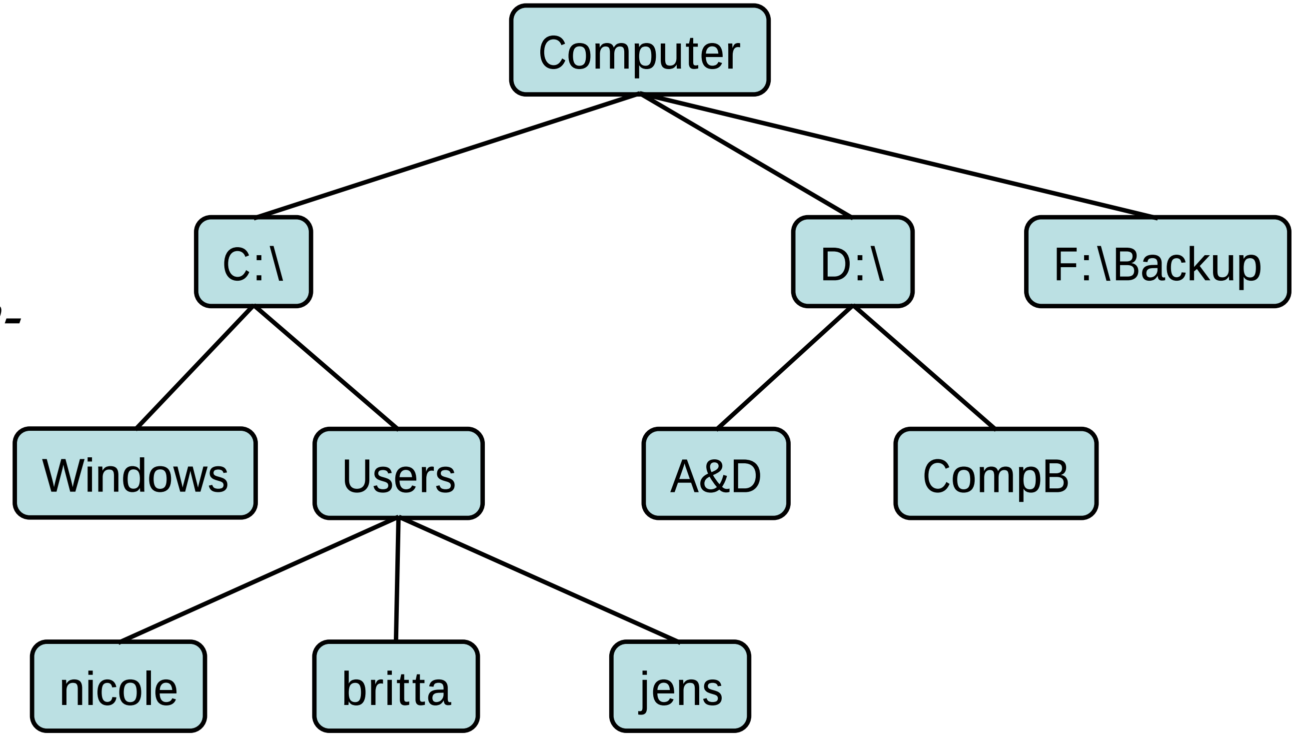
Was ist ein (Informations-) Baum?

In der Informatik repräsentieren Bäume abstrakte, *hierarchische* Strukturen.

Ein Baum besteht aus Knoten, welche in *Eltern-Kind Relation* stehen.

Anwendungen:

- *Organigramm*
- *Dateisystem*
- *Programmierungs-Umgebungen*
- *...*



Baum Terminologie

Wurzel (Root):
Knoten ohne Elternknoten *(A)*

Interner Knoten:
Knoten mit min. einem Kind *(A, B, C, F)*

Externer Knoten (Blatt):
Knoten ohne Kinder *(E, I, J, K, G, H, D)*

Vorgängerknoten:
Eltern, Grosseltern, etc.

Tiefe eines Knotens**:**
Anzahl Vorgänger

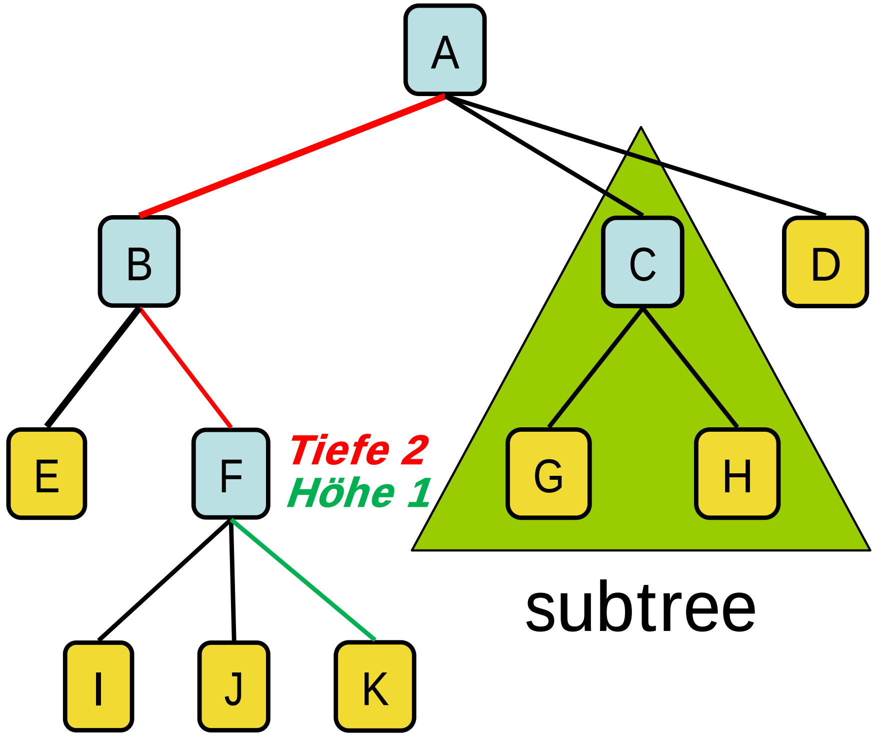
Höhe eines *Knoten (rekursive Definition)* :
Externer Knoten: 0
Interner Knoten: 1 + **maximale** Höhe aller Nachfolgerknoten

Höhe eines Baums: *Höhe der Wurzel*

Nachfolger eines Knotens:
Kind, Grosskind, etc.

Subtree (Unterbaum) : Baum aus einem Knoten und seinen Nachfolgern

Sibling: Zwillingsknoten



Tree ADT

Die ***Position*** dient in dem *Tree ADT* (Abstrakter Datentyp) als Abstraktion für *Knoten*.
Zugriff auf das Element: ***Element Position.getElement()***

Zugriffsmethoden:

- *Position* ***root****()*
- *Position* ***parent(p****)*
- *PositionList* ***children(p****)*
- *integer* ***numCildren(p****)*

Hilfsmethoden:

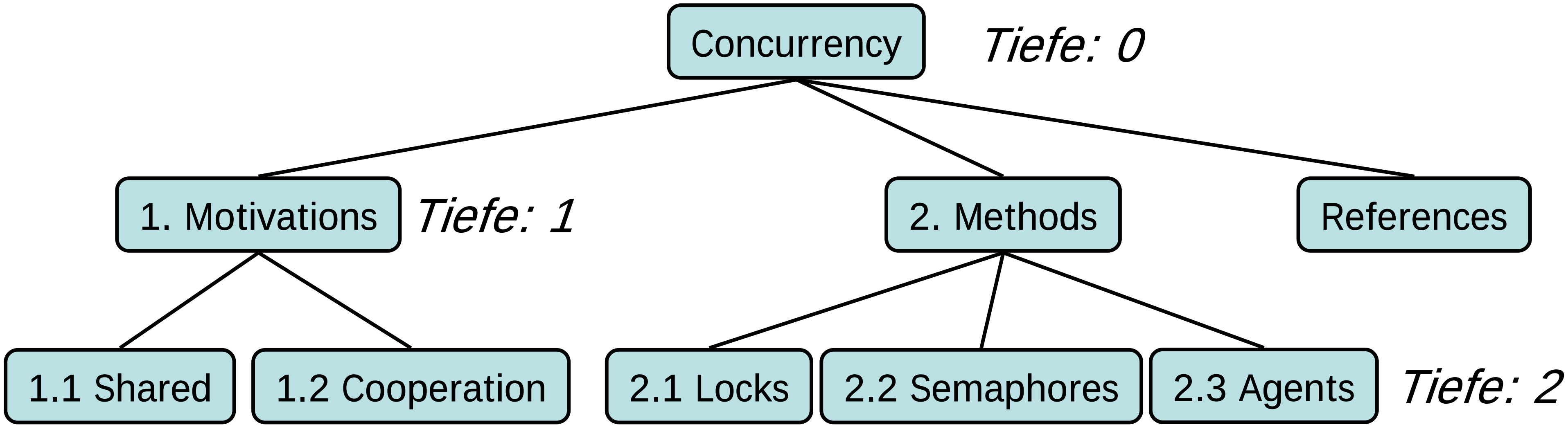
- *integer* ***size****()*
- *boolean* ***isEmpty****()*
- *Iterator* ***iterator****()*

Abfragemethoden:

- *boolean* ***isInternal(p****)*
- *boolean* ***isExternal(p****)*
- *boolean* ***isRoot(p****)*

Die Tiefe

- Sei ***v*** ein Knoten des Baumes T.
- Dann ist die ***Tiefe*** von ***v*** definiert als die ***Anzahl Vorgänger*** von ***v*** (ohne ***v*** selbst!), also die Anzahl Kanten bis zur Wurzel.
  (Anzahl der Knoten auf dem Pfad von der Wurzel zum Knoten ***v***)
- Beispiel:
  - Falls ***v*** die Wurzel ist, dann ist die Tiefe von ***v*** : 0
  - Falls ***v*** nicht die Wurzel ist, dann ist die Tiefe von ***v*** :
    1 + Tiefe des Eltern-Knoten von ***v***

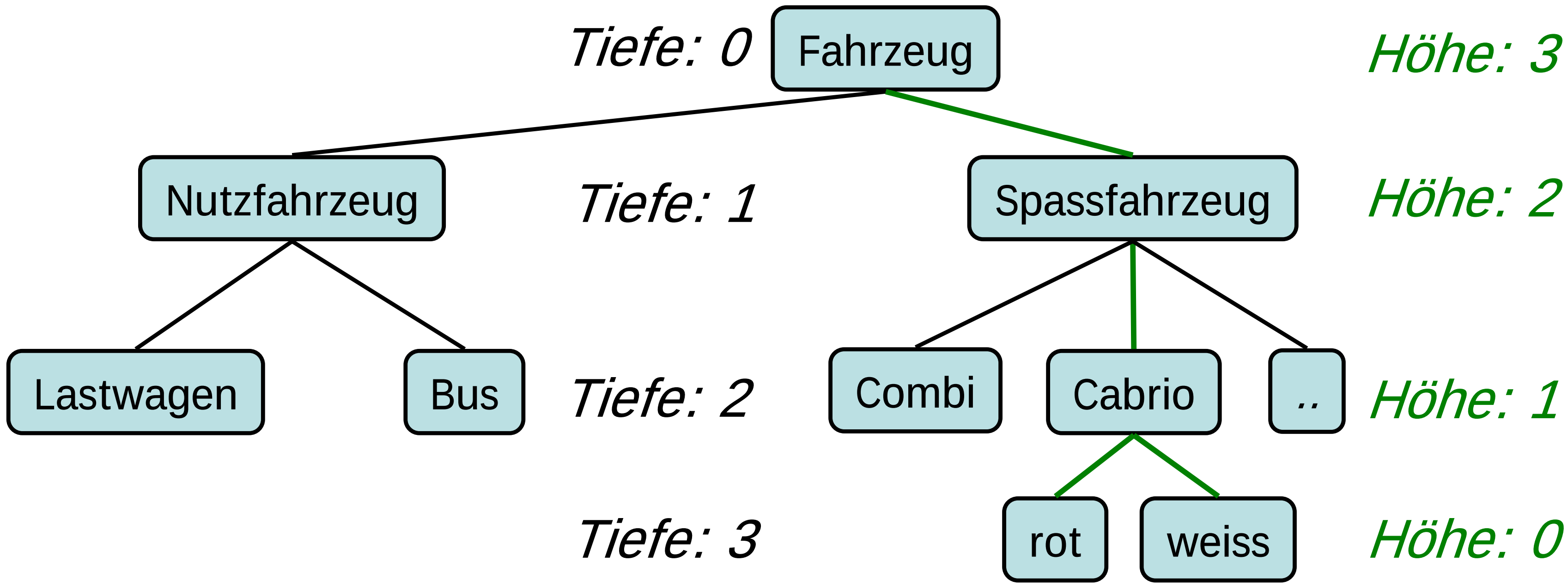


Tiefe: Implementation

```
Algorithm depth(T, v)
  if v is the root of T then
    return 0
  else
    return 1 + depth(T, v.parent())
```

Die Höhe

- Sei ***v*** ein Knoten des Baumes T.
- Dann ist die ***Höhe des Teilbaumes mit Wurzel v*** definiert als die ***maximale Tiefe des Teilbaumes*** mit ***Wurzel v***.
  (Die Höhe eines Baumes ist gleich der Tiefe des tiefsten Knotens im Baum)
- Anschaulich:
  - Die Höhe gibt die Anzahl Ebenen des Baumes an.



Höhe: Implementation

Höhe eines *Knoten* : rekursiv
Externer Knoten: 0
Interner Knoten: 1 + **maximale** Höhe aller Nachfolgerknoten

```
Algorithm height(T, v)
  h = 0
  for all children w of v
    h = max(h, 1 + height(T ,w))
  return  h
```

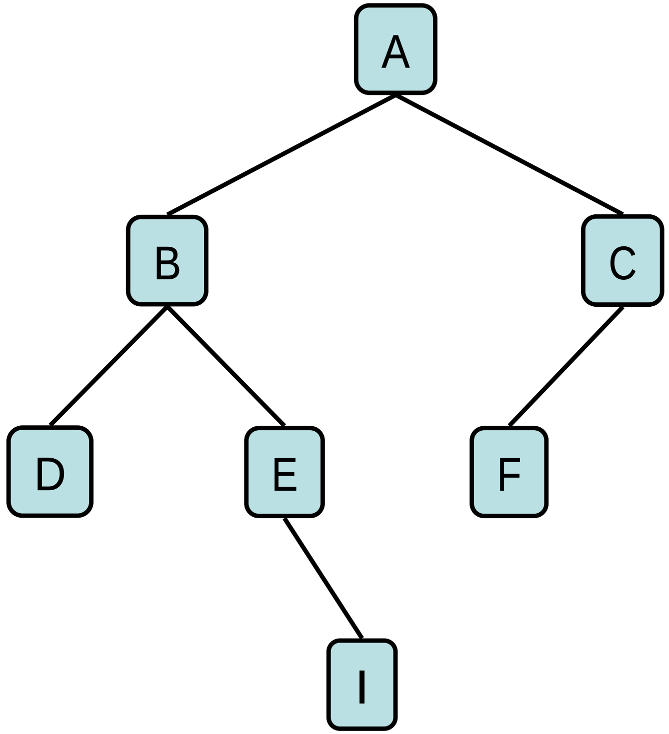
Binäre Bäume

- Ein binärer Baum ist ein Baum mit folgenden Eigenschaften:
  1. Jeder interne Knoten besitzt höchstens zwei Kinder
     (*exakt* zwei bei *echten Binärbäumen*).
  2. Die Kinder eines Knotens sind ein geordnetes Paar (links, rechts).

- Die Kinder eines internen Knotens werden als linkes und rechtes Kind bezeichnet.

- Alternative **rekursive Definition**:
  Ein binärer Baum ist entweder:
  - ein Baum, bestehend aus keinem oder einem einzelnen Knoten, *oder*
  - ein Baum, dessen Wurzel ein geordnetes Paar Kinder besitzt, welche selber wieder binäre Bäume sind.

Anwendungen, z.B.:
- arithmetische Ausdrücke
- Entscheidungsprozesse
- Suchen

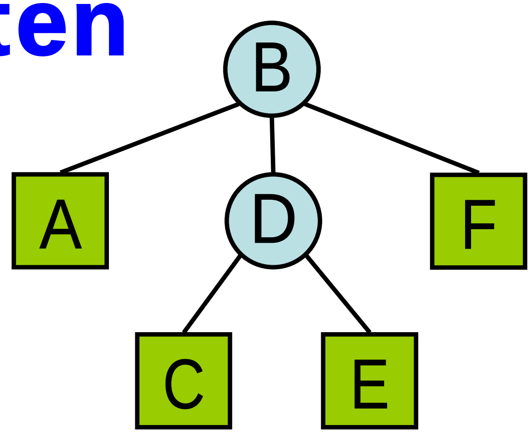


Binärbaum ADT

- Der ***BinaryTree ADT*** erweitert den *Tree ADT*, d.h. er erbt alle Methoden des Tree ADT.

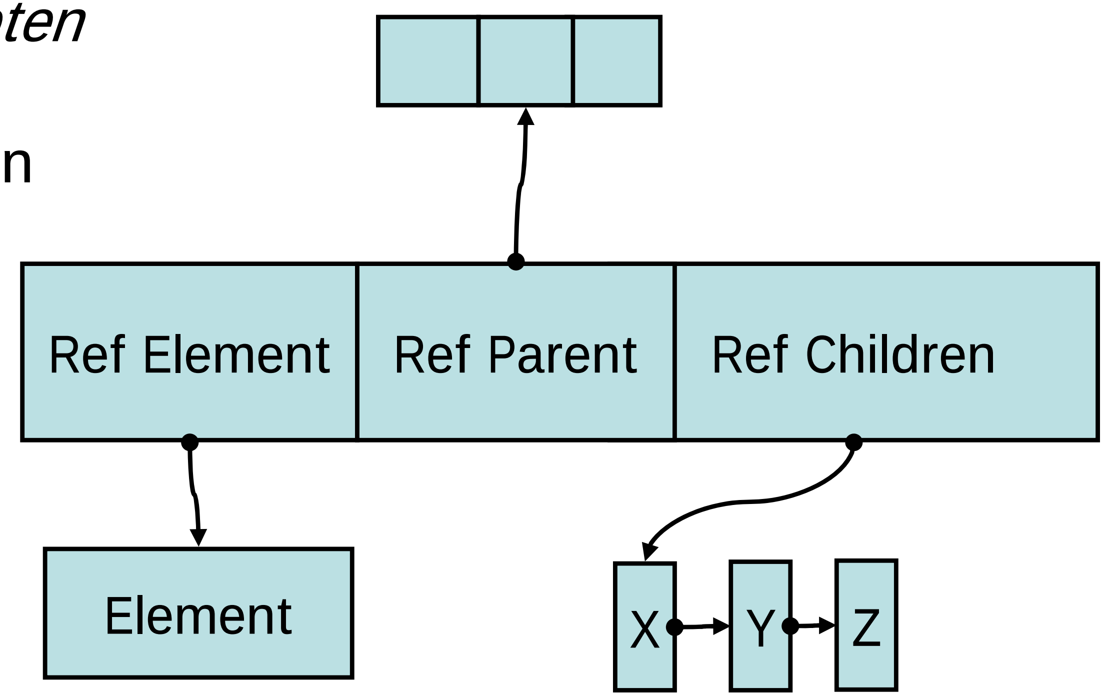
- Zusätzliche Methoden:
  - *Position* ***left(p)***
  - *Position* ***right(p)***
  - *Position* ***sibling(p)***

Speicherverfahren für Bäume: Verlinkte Baum-Knoten

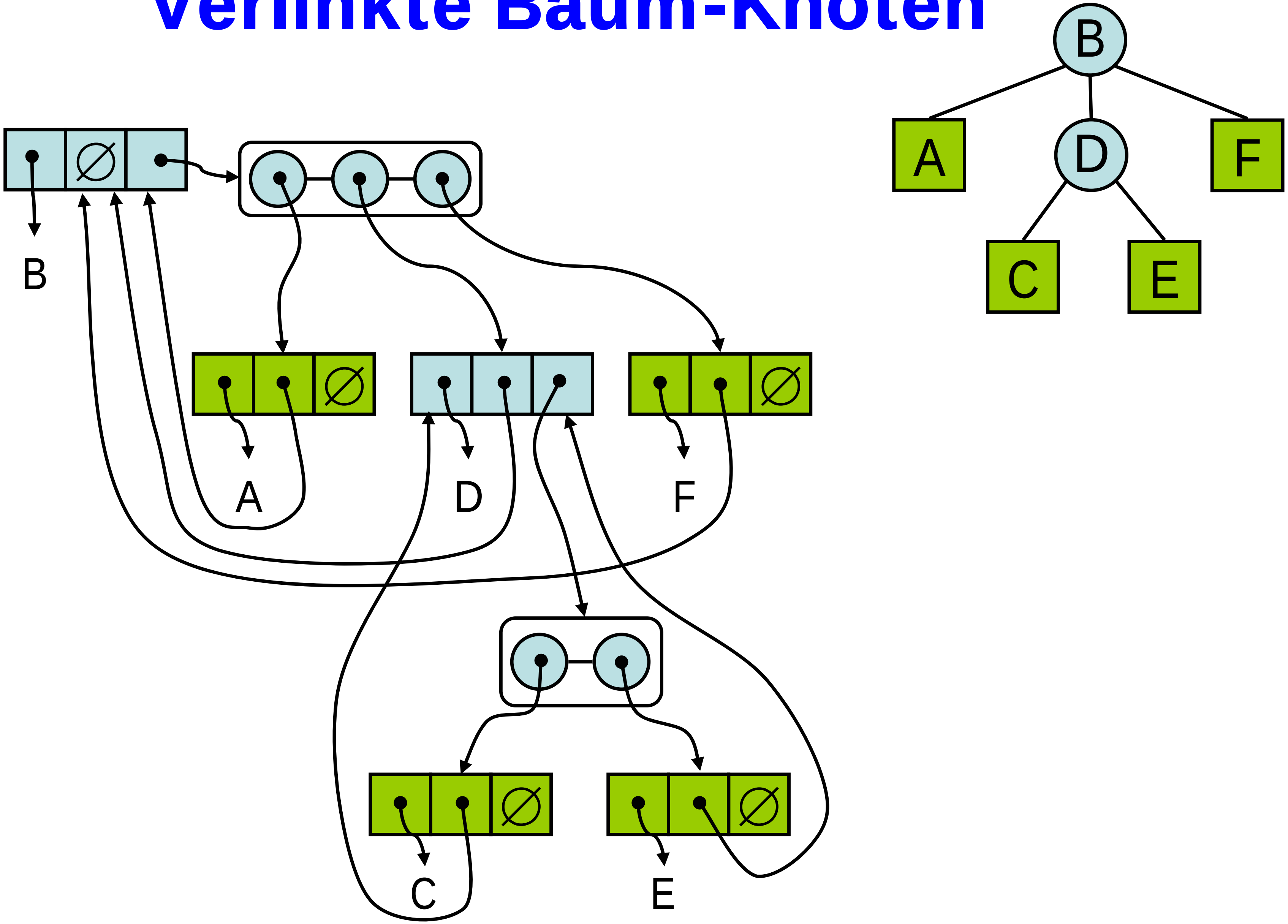


- Ein Baumknoten wird repräsentiert durch ein Objekt mit folgendem Aufbau:
  - *Element*
  - *Elternknoten*
  - *Sequence mit Kindknoten*

- Die Knoten implementieren den Position ADT



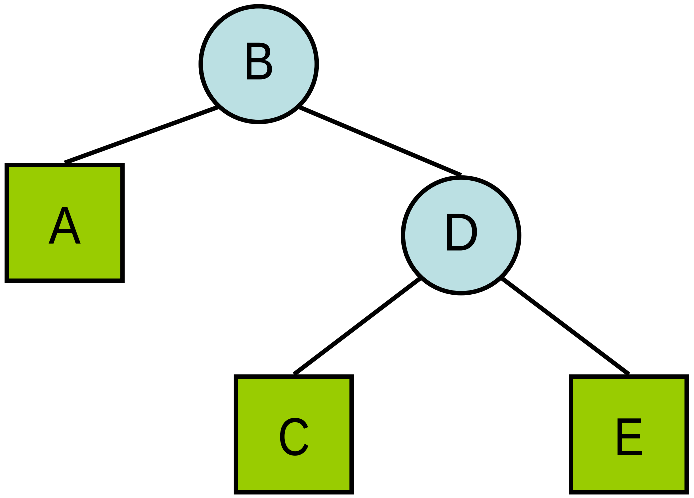
Speicherverfahren für Bäume: Verlinkte Baum-Knoten

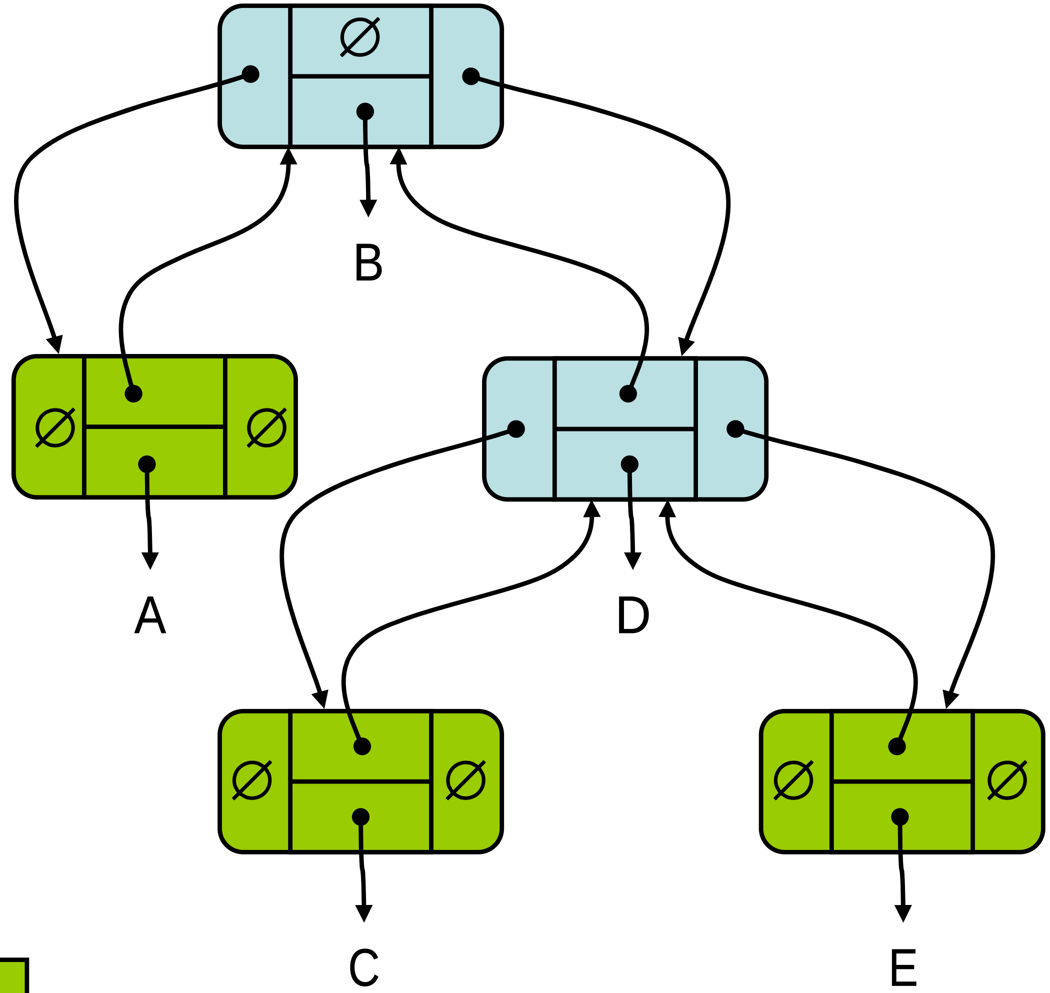


Verlinkte Baum-Knoten für Binary-Trees

- In diesem Fall kann man auf die Sequence verzichten:
  - *Element*
  - *Eltern Knoten*
  - *Linker Kind Knoten*
  - *Rechter Kind Knoten*

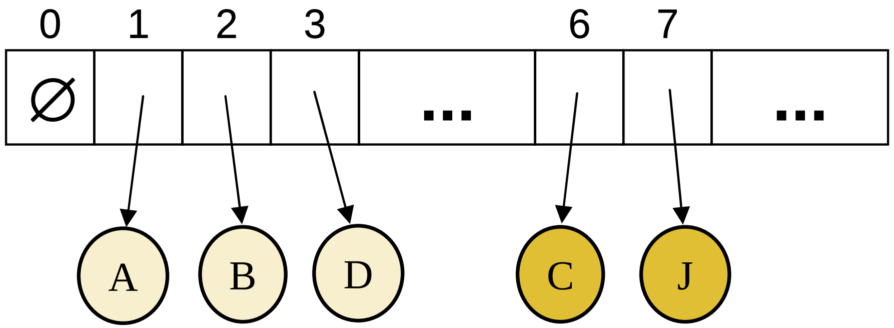
- Auch hier implementieren die Knoten den Position ADT

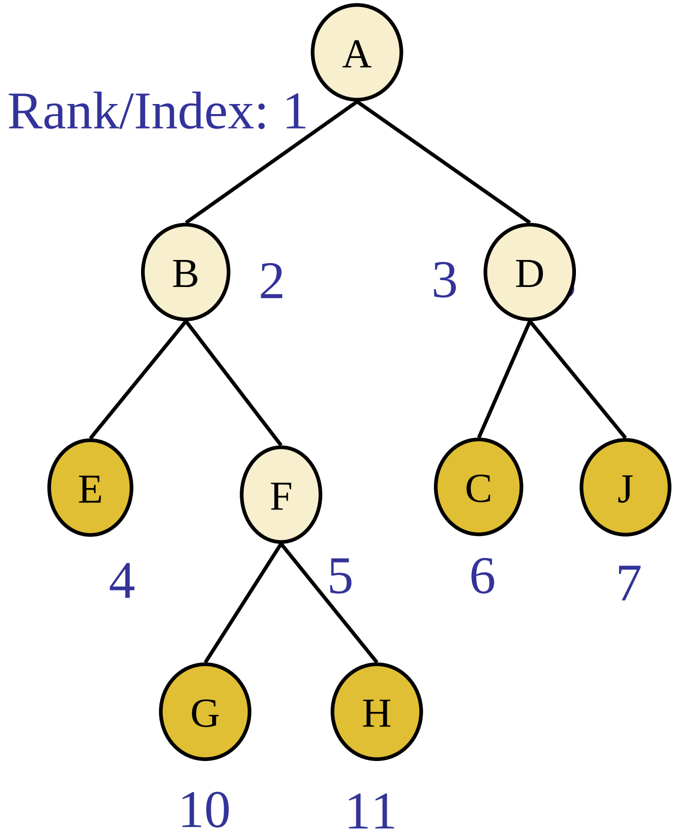




Speicherverfahren für Bäume: Array basiert

Die Knoten werden in einem Array gespeichert (Beispiel für Binär-Baum):

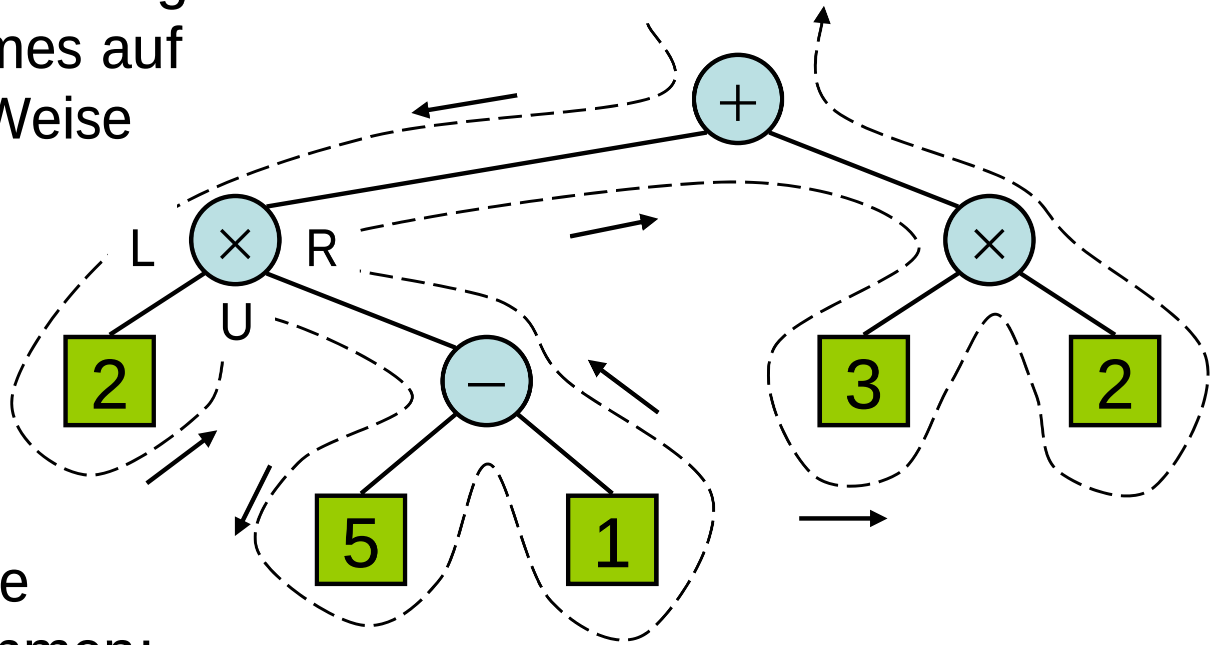




- sei **rank(node)** folgendermassen definiert:
  - rank(root) = 1
  - Falls node linkes Kind des parent(node): rank(node) = 2 * rank(parent(node))
  - if node is the right child of parent(node): rank(node) = 2 * rank(parent(node))+1

Baum-Traversierungen

- Bei einer Baum-Traversierung werden alle Knoten eines Baumes auf systematisch Art und Weise besucht werden.



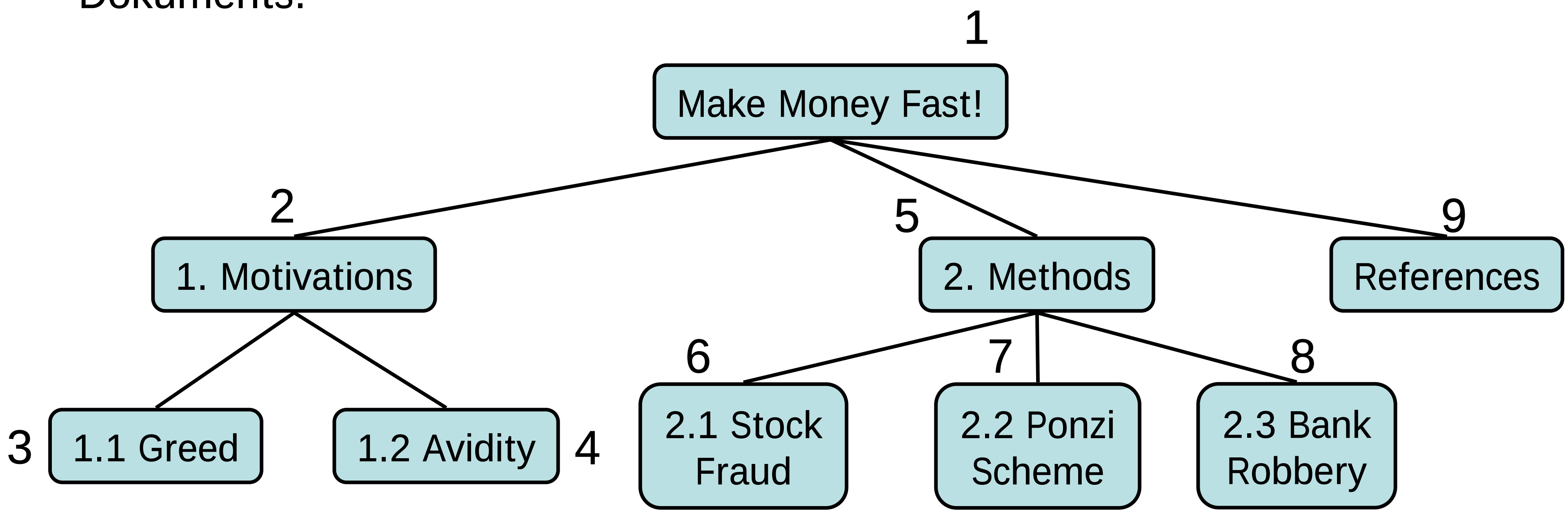
- Es gibt unterschiedliche Traversierungs-Algorithmen:
  - *Preorder*
  - *Postorder*
  - *Breadth-First*
  - *Inorder*

Preorder Traversierung

- In einer *Pre-Order Traversierung* wird ein Knoten ***vor*** seinen Nachfolgern besucht.

```
Algorithm preOrder(v)
  visit(v)
  for each child w of v
    preOrder (w)
```

- Beispiel:
  Drucken eines strukturierten Dokuments.

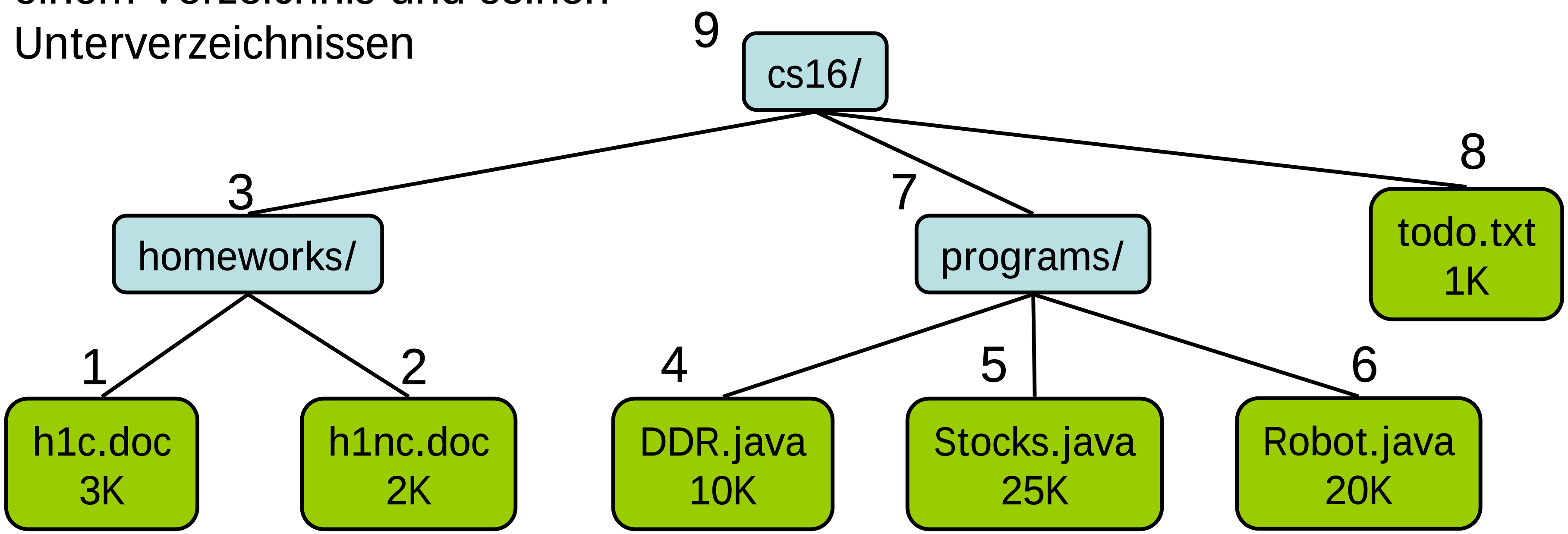


Postorder Traversierung

- In einer *Post-Order Traversierung* wird ein Knoten ***nach*** seinen Nachfolgern besucht.

```
Algorithm postOrder(v)
  for each child w of v
    postOrder (w)
  visit(v)
```

- Beispiel:
  Angabe des benutzten Speichers in einem Verzeichnis und seinen Unterverzeichnissen

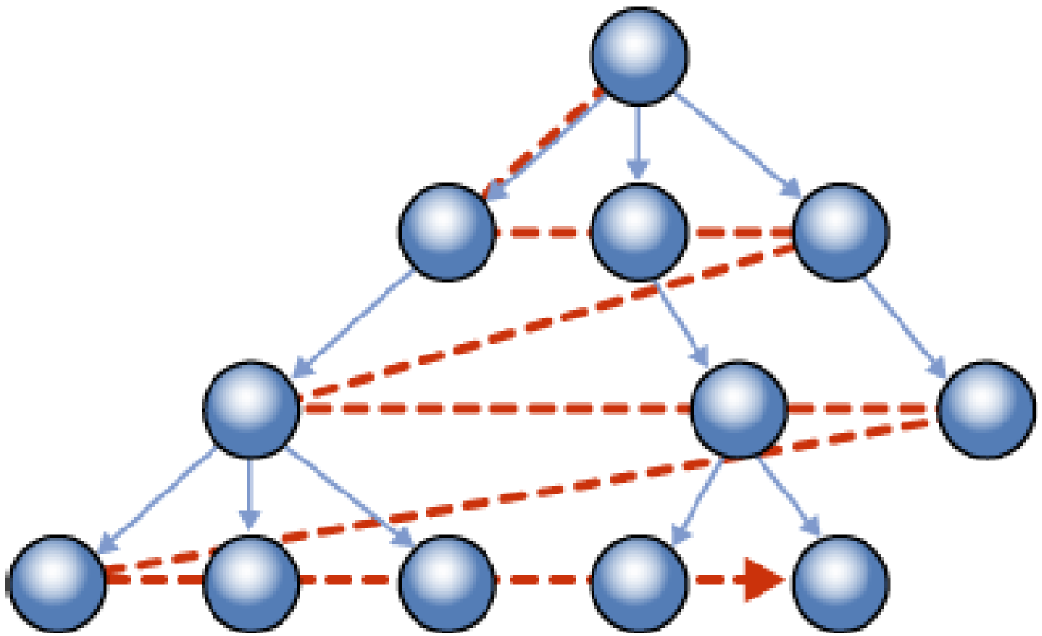


Breadth-First Traversierung

- In einer *Breadth-First Traversierung* werden zuerst alle Knoten einer Tiefe ***t*** besucht, bevor the Knoten der Tiefe ***t+1*** besucht werden.

```
Algorithm breadthFirst()
  Initialize queue Q containing root
  while Q not empty do
    v = Q.dequeue()
    visit(v)
    for each child w in children(v) do
      Q.enqueue(w)
```

- Beispiel:

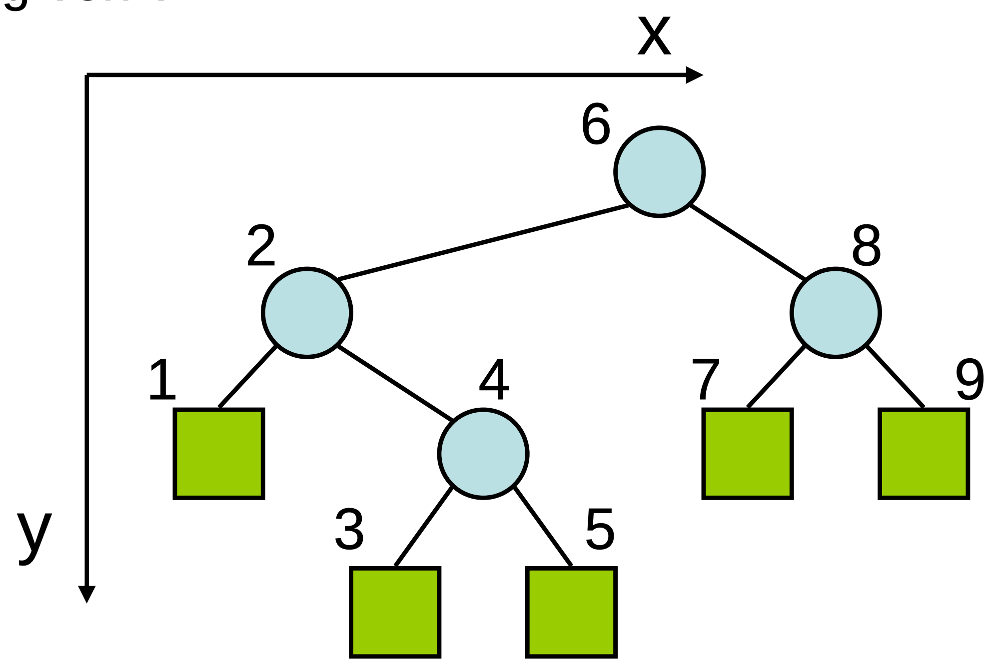


Inorder Traversierung in Binärbäumen

- In einer *Inorder Traversierung* wird ein Knoten ***nach*** seinem *linken Subtree* und ***vor*** seinem *rechten Subtree* besucht.

- Beispiel:
  Darstellung von binären Bäumen
  - x(v) = inorder Rang von v
  - y(v) = Tiefe von v

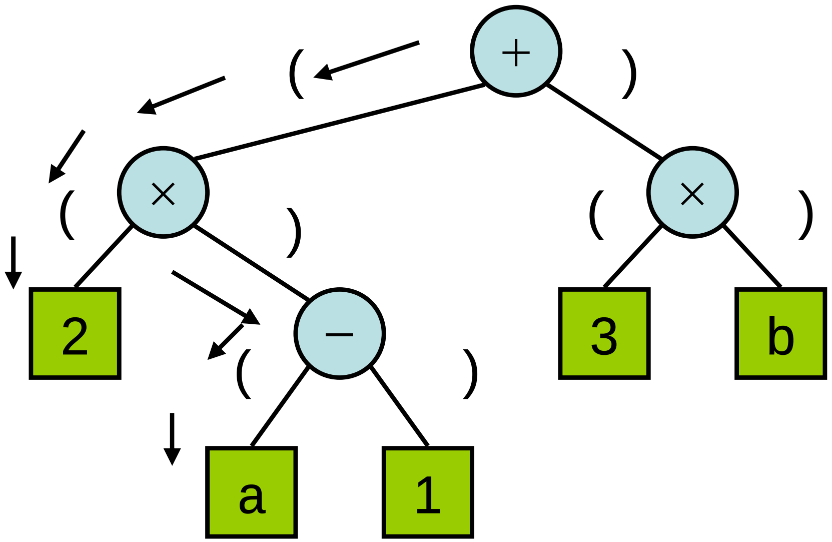
```
Algorithm inOrder(v)
   if hasLeft(v)
      inOrder(left(v))
   visit(v)
   if hasRight(v)
      inOrder(right(v))
```



Ausgabe Arithmetischer Ausdrücke

- Spezialisierung einer *Inorder Traversierung*
  - print "(" **vor** der Traversierung des linken Subbaums
  - Ausgabe des Operanden resp. Operators beim Besuch des Knotens
  - print ")" **nach** der Traversierung des rechten Subbaums

```
Algorithm printExpression(v)
  if hasLeft(v)
    print("(")
    printExpression(left(v))
  print(v.element())
  if hasRight(v)
    printExpression(right(v))
    print (")")
```



Resultat: $((2 \times (a - 1)) + (3 \times b))$

Arithmetische Ausdrücke auswerten

- Spezialisierung einer *Postorder Traversierung*
  - rekursive Methode, welche den Wert eines Unterbaums liefert
  - Beim Besuch eines internen Knotens werden die Werte des Subbaumes kombiniert / der Subbaum ausgewertet

```
Algorithmus evalExpr(v)
  if isExternal (v)
    return v.element ()
  else
    x ← evalExpr(leftChild (v))
    y ← evalExpr(rightChild (v))
    ◊ ← Operator bei v
    return x ◊ y
```