

# Bäume und Graphen

In diesem Kapitel behandeln wir erste Algorithmen mit dynamischen Strukturen, wie Bäume und Graphen.

## Inhalt

1. Bäume .....	2
1.1. Grundlagen .....	2
1.2. Repräsentation von Binärbäumen .....	9
1.2.1. Arrays .....	9
1.2.2. Zeiger .....	11
1.3. Operationen auf Binärbäumen .....	27
1.3.1. Kopieren von Bäumen .....	27
1.3.2. Äquivalenz von Bäumen .....	28
1.4. Repräsentation von Bäumen durch Binärbäume .....	29

# 1. Bäume

Verkettete Listen haben neben der Nutzinformation einen Zeiger auf den Nachfolger als Verwaltungsinformation.

Will man hierarchische Strukturen realisieren, braucht man mehr Verwaltungsinformation. Wir betrachten Bäume.

## 1.1. Grundlagen

Zunächst werden wir einige Begriffe kennen lernen.

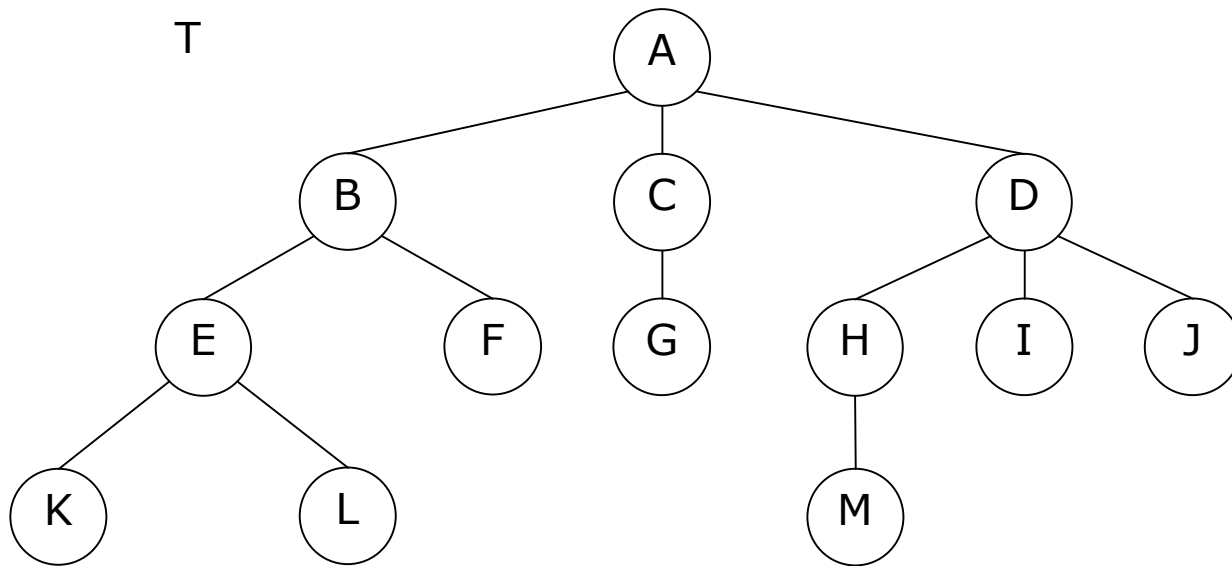
### Definition 1-1 (Baum)

Ein *Baum* (engl. tree) besteht aus einer endlichen Menge von Knoten (engl. node) so dass gilt:

- (i) ein ausgezeichnete Knoten ist die *Wurzel*;
- (ii) die restlichen Knoten sind partitioniert in  $n \geq 0$  disjunkte Mengen  $T_1, T_2, \dots, T_n$  wobei jeder dieser Mengen selbst ein Baum ist.  
 $T_1, T_2, \dots, T_n$  nennt man *Unterbäume* der Wurzel.

Die Bedingung, dass  $T_1, T_2, \dots, T_n$  **disjunkte** Mengen sind, verbietet es, dass zwei Unterbäume (außer über die Wurzel) miteinander verbunden sind. Weiterhin folgt, dass jeder Knoten im Baum von der Wurzel aus erreichbar.

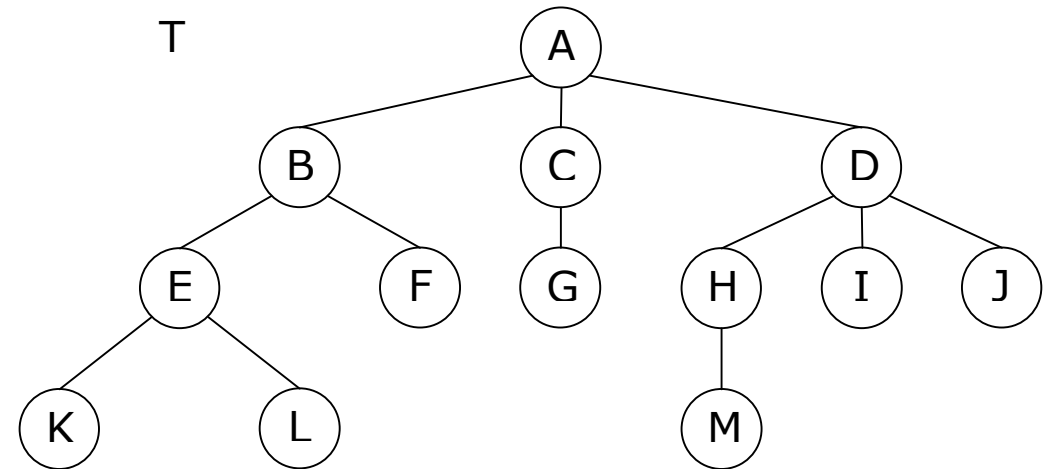
Beispiel:



Folgende Begriffe tauchen immer wieder auf:

- Der **Grad** (engl. *degree*) **eines Knotens** ist die Anzahl seiner Teilbäume. (z.B. Grad von B ist 2)
- Der **Grad eines Baums** ist der maximale Grad seiner Knoten. (Grad von T ist 3)
- Ein **Blatt** ist ein Knoten mit Grad 0. (K, L, F, G, M, I, J)
- Die Wurzel eines Unterbaums eines Knotens  $X$  nennt man *Kind* von  $X$ .  $X$  ist dann der **Vater** (oder Elternteil, engl. *parent*) des **Kindes**.
- Kinder mit dem selben Vater nennt man **Geschwister** (eng. *sibling*)

- Die **Vorfahren** (engl. *ancestors*) eines Knoten sind die Knoten auf dem Pfad von der Wurzel zu dem Knoten.
- Der **Level** eines **Knotens** ist rekursiv definiert als:  
Level der Wurzel ist 1; wenn ein Knoten den Level  $i$  hat, haben die Kinder den Level  $i+1$ .  
(z.B. Level von D ist 2)
- Die **Höhe** oder Tiefe eines **Baums** ist definiert als maximaler Level aller Knoten des Baums. (Level von T ist 4)



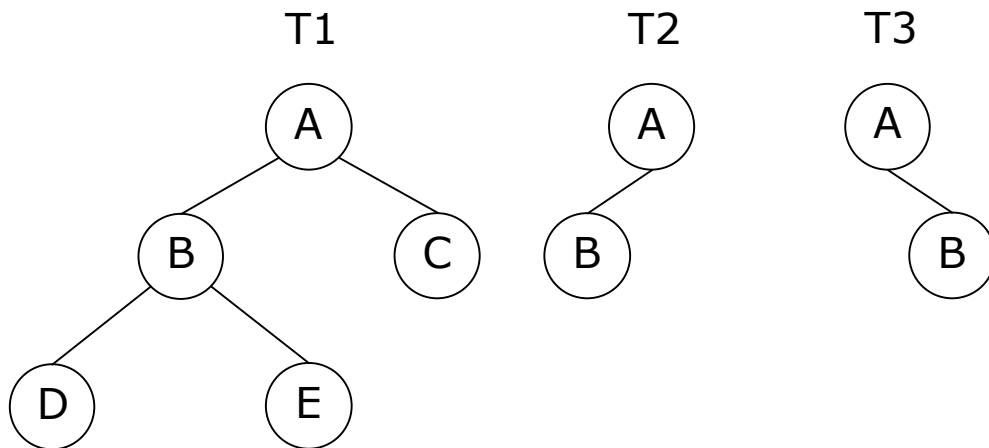
Um Bäume zu implementieren braucht man variable Anzahl von Verweisen auf die Kinder, eben in Abhängigkeit des Grades eines Knotens.

Einfacher ist es, wenn man Bäume mit fixem Grad betrachtet. Wir werden zunächst binäre Bäume diskutieren, das sind Bäume, bei denen der Grad jeden Knotens kleiner oder gleich 2 ist.

## Definition 1-2 (Binärbaum)

Ein *Binärbaum* ist eine Menge von Knoten, die entweder leer ist oder aus einer Wurzel und zwei disjunkten Binärbäumen besteht, dem *linken* und *rechten Unterbaum*.

Beispiele (drei unterschiedliche Binärbäume)



Folgende Begriffe tauchen bei Binärbäumen immer wieder auf:

- T2 und T3 sind **entartete** Binärbäume (links- bzw. rechtsentartet). Jeder Knoten hat 0 oder 1 Kind)
- T2 ist ein **vollständiger** Binärbaum (jeder Knoten hat 0 oder 2 Kinder).

Zur Analyse von Algorithmen auf Binärbäumen werden wir zunächst noch einige **Sätze** herleiten. Sie zeigen **Eigenschaften von Binärbäumen**.

## Satz 1-1:

Die maximale Anzahl von Knoten auf Level  $i$  eines Binärbaums ist  $2^{i-1}$  für  $i \geq 1$ .

## Beweis 1-1

Der Beweis wird durch Induktion über die Levelzahl  $i$  geführt.

### Induktionsbasis ( $i=1$ ):

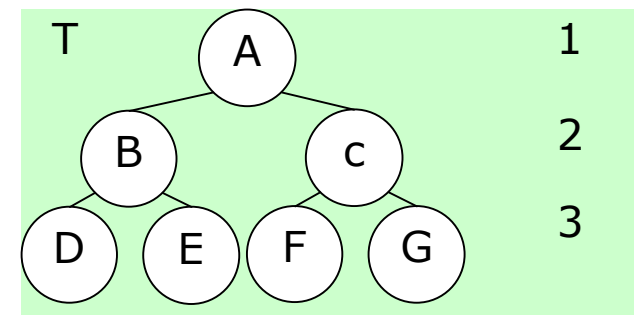
Die Wurzel ist der einzige Knoten auf Level  $i=1$ . Daher ist die maximale Anzahl von Knoten auf Level  $i = 1 = 2^0 = 2^{i-1}$ .

### Hypothese

Für alle  $j$ ,  $1 \leq j < i$  ist die maximale Knotenanzahl auf Level  $j$  gleich  $2^{j-1}$ .

### Induktionsschritt ( $i-1 \rightarrow i$ )

Die maximale Knotenanzahl auf Level  $i-1$  ist gleich  $2^{i-2}$  gilt wegen der Hypothese. Da jeder Knoten in einem Binärbaum höchstens Grad 2 haben kann ist die maximale Knotenanzahl auf Level  $i$

$$\begin{aligned} &= 2 * \text{Maximalanzahl auf Level } i-1 \\ &= 2 * 2^{i-2} \\ &= 2^{i-1}. \end{aligned}$$


## Satz 1-2

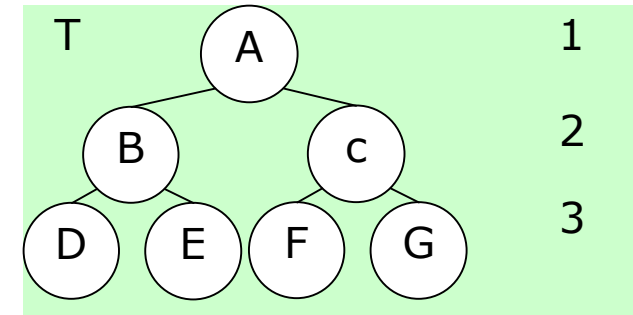
Die *maximale Anzahl von Knoten* in einem Binärbaum der Tiefe  $k$  ist  $2^k - 1$ ,  $k \geq 1$ .

## Beweis 1-2

Die maximale Knotenanzahl  $K$  eines Binärbaums der Tiefe  $k$  wird einfach summiert über alle Level.

Sei  $m_i$  die maximale Knotenanzahl auf Level  $i$ .

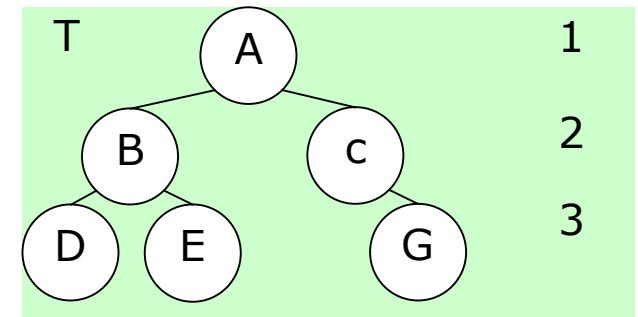
$$\text{Dann ist } K = \sum_{i=1}^k m_i = \sum_{i=1}^k 2^{i-1} = 2^k - 1.$$



Ein weiterer Satz macht Aussagen über das **Verhältnis von Gesamtanzahl von Knoten** und **Blättern** in einem Binärbaum.

## Satz 1-3

Sei  $n_0$  die Anzahl der Blätter und  $n_2$  die Anzahl der Knoten mit Grad 2 in einem Binärbaum. Es gilt:  $n_0 = n_2 + 1$ .



### Beweis 1-3

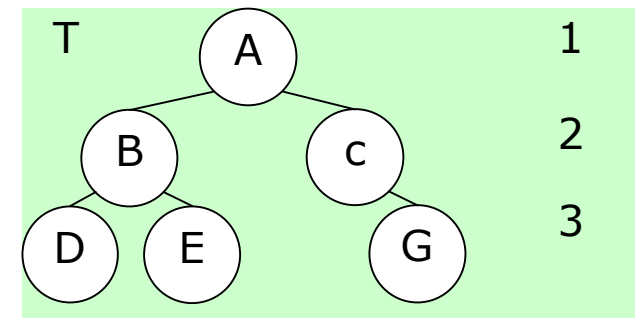
Sei  $n_i$  die Anzahl der Knoten mit Grad  $i$  ( $0 \leq i \leq 2$ ) und  $n$  die Gesamtzahl der Knoten in  $T$ . Da  $T$  Binärbaum ist gilt:

$$(a) \quad n = n_0 + n_1 + n_2.$$

Wenn man die Äste des Baums betrachtet, so sieht man, dass alle Knoten außer der Wurzel einen Ast vom Vater zum Knoten haben. Sei  $B$  die Anzahl der Äste, dann gilt  $n = B + 1$ . Alle Äste gehen entweder von einem Knoten mit Grad 1 oder 2 aus. Daher gilt:  $B = n_1 + 2n_2$  und wir erhalten:

$$(b) \quad n = B + 1 \\ = n_1 + 2n_2 + 1$$

Wenn man (b) und (a) gleich setzt erhält man:  
 $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$  und somit  $n_0 = n_2 + 1$ .



Insgesamt haben wir gesehen, dass Eigenschaften von Strukturen mathematisch formulierbar und beweisbar sind.

## 1.2.Repräsentation von Binärbäumen

Hier werden zwei Arten der Implementierung von Bäumen gezeigt:

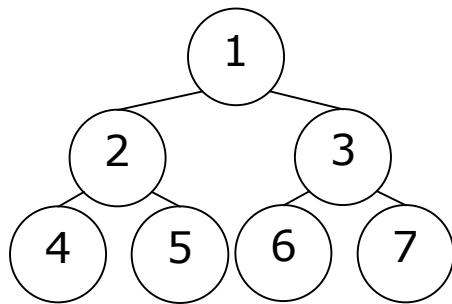
- Arrays (Sie würden jetzt natürlich Vektoren der STL verwenden!)
- Zeiger

### 1.2.1. Arrays

Eine elegante Methode zum Abspeichern von vollständigen Bäumen ist es, Arrays zu verwenden und die **Knoten sequentiell zu nummerieren**: man startet mit den Knoten auf Level 1, dann werden die Knoten auf Level 2 genommen usw; dabei werden die **Knoten** immer von **links nach rechts** hoch gezählt.

Der **Knoten  $i$**  wird dann im Array an **Position  $i$**  abgespeichert:

Nummerierung



Array

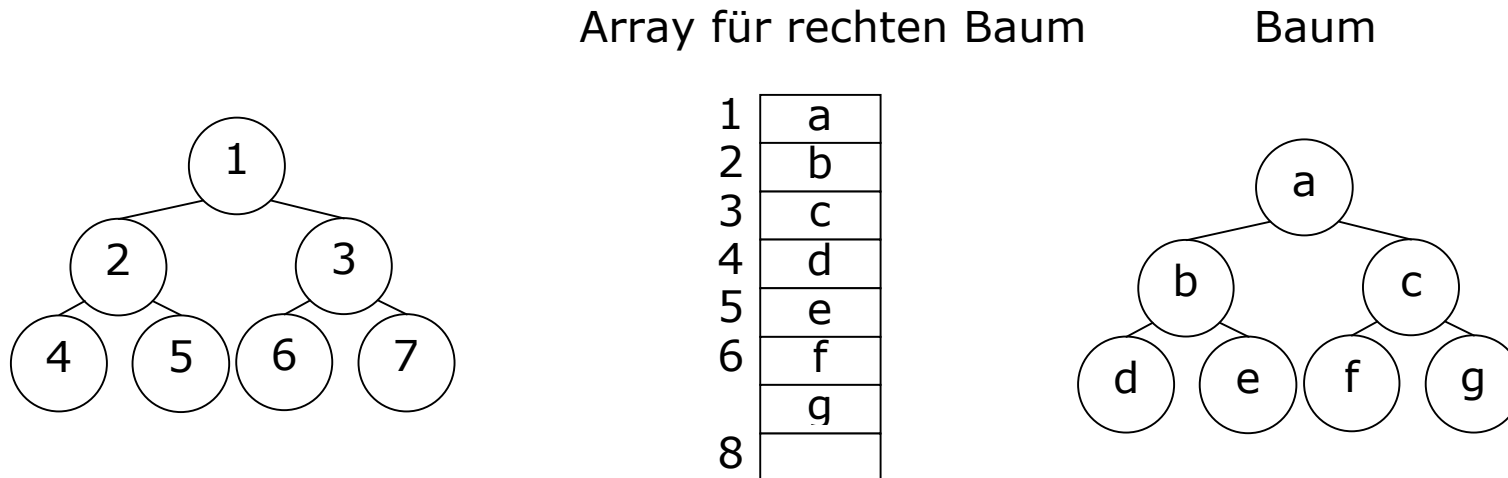
1	a
2	b
3	c
4	d
5	e
6	f
7	g
8	

Ein Knoten in dieser Darstellung kann dann gefunden werden durch Ausnutzung des folgenden Satzes:

### Satz 1-4

Wird ein vollständiger Baum mit  $n$  Knoten sequentiell representiert im Array  $T$ , dann gilt für jeden Knoten mit Index  $i$ ,  $1 \leq i \leq n$  :

- (i)  $parent(i) = T[i / 2]$  wenn  $i \neq 1$  (Wurzel hat keinen Vater)
- (ii)  $leftChild(i) = T[2i]$ , wenn  $2i \leq n$ . Ist  $2i > n$ , dann hat  $i$  kein linkes Kind.
- (iii)  $rightChild(i) = T[2i + 1]$ , wenn  $2i + 1 \leq n$ . Ist  $2i + 1 > n$ , dann hat  $i$  kein rechtes Kind.



### Beweis 1-4

zu Hause, wer Spaß daran hat.

## Hörsaalübung:

Entwickeln Sie eine Klasse Tree, die einen Binärbaum wie o.a. speichert.

Die Methoden sollen sein:

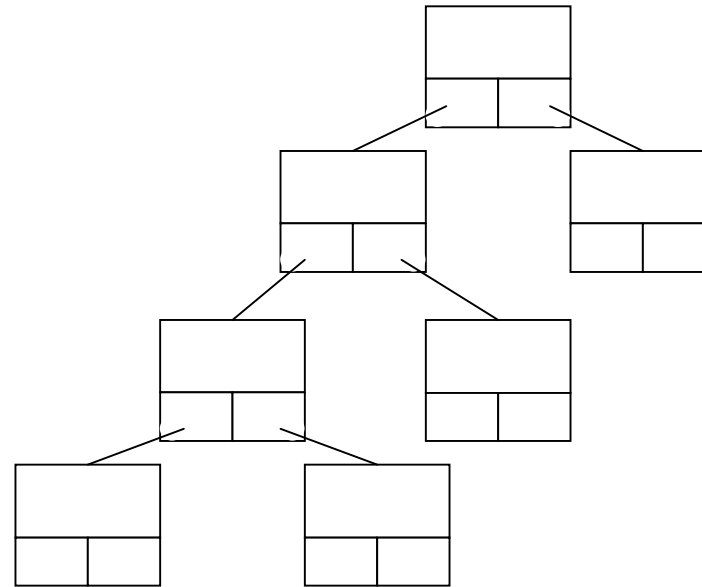
- `int getLeftChild(int i),`
- `int getRightChild(int i),`
- `int getParent(int i),`
- `printNode(int i),`
- im Konstruktor soll der Baum initialisiert werden.

## 1.2.2. Zeiger

Eine Methode, Binärbäume dynamisch zu speichern ist es eine rekursive Datenstruktur zu verwenden.

Ein Baumknoten wird durch die folgende Struktur abgebildet:

```
struct baum {  
    int info;  
    struct baum *links;  
    struct baum *rechts;  
}
```

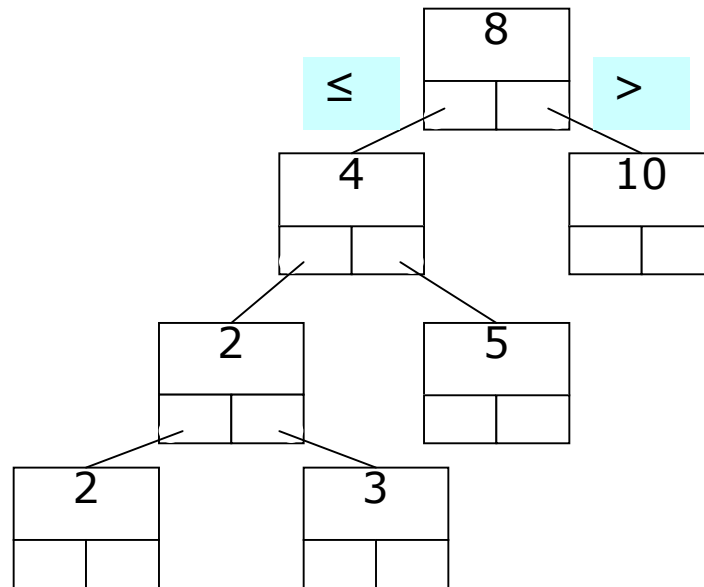


Binärbäume werden z.B. verwendet, um eine Informationsmenge zu speichern und sortiert ausgeben zu können. Wenn die Speicherung geschickt gewählt wird, kann man zudem effizient nach Elementen suchen.

### Definition 1-3 (binärer Suchbaum)

Sei  $T$  ein Binärbaum mit  $w$  als Wurzenknoten.  $T$  heißt *binärer Suchbaum*, wenn gilt:

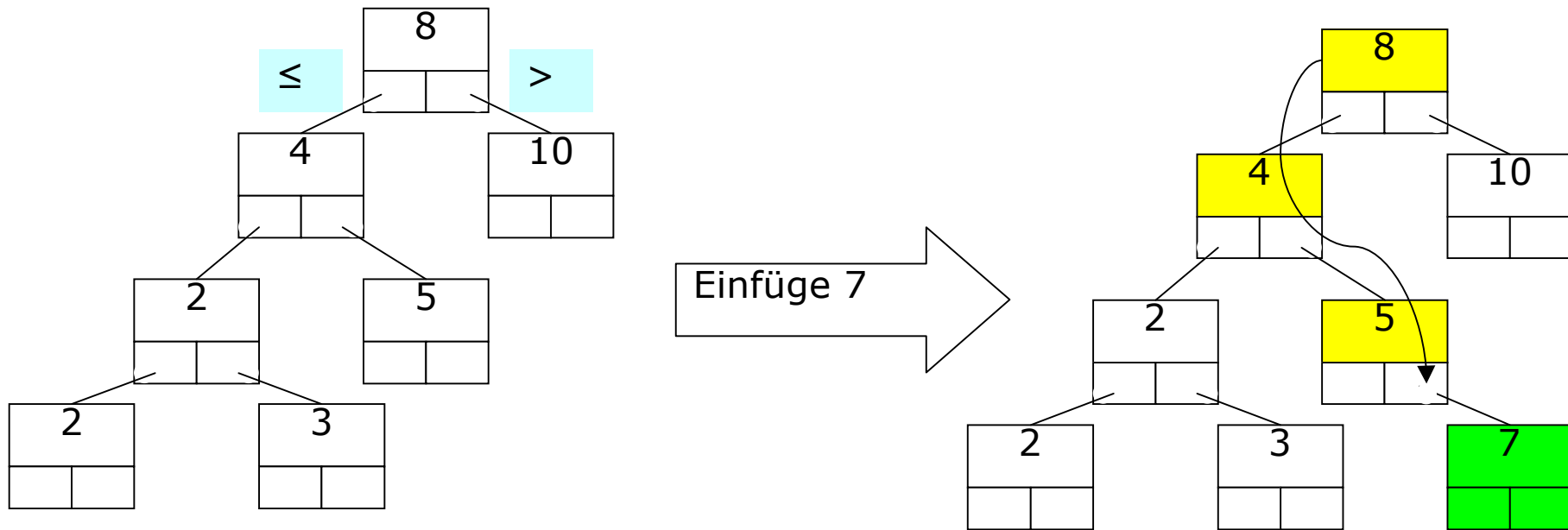
- (i) alle Knoten im linken Unterbaum haben kleinere Werte als  $w$  oder sind gleich dem Wert von  $w$ ,
- (ii) alle Knoten im rechten Unterbaum haben größere Werte als  $w$ .



Das **Einfügen** eines neuen **Knotens** erfolgt so, dass er immer an ein **Blatt** an der richtigen Stelle angehängt wird:

Man lässt den Knoten von der Wurzel aus „durchfallen“, bis eine Blattposition gefunden ist.

Beispiel:



Die Funktion zum Einfügen ist rekursiv implementiert – hier sieht man die Stärke von rekursiven Datenstrukturen und rekursiven Verfahren.

## Programm (Einfügen in binären Suchbaum)

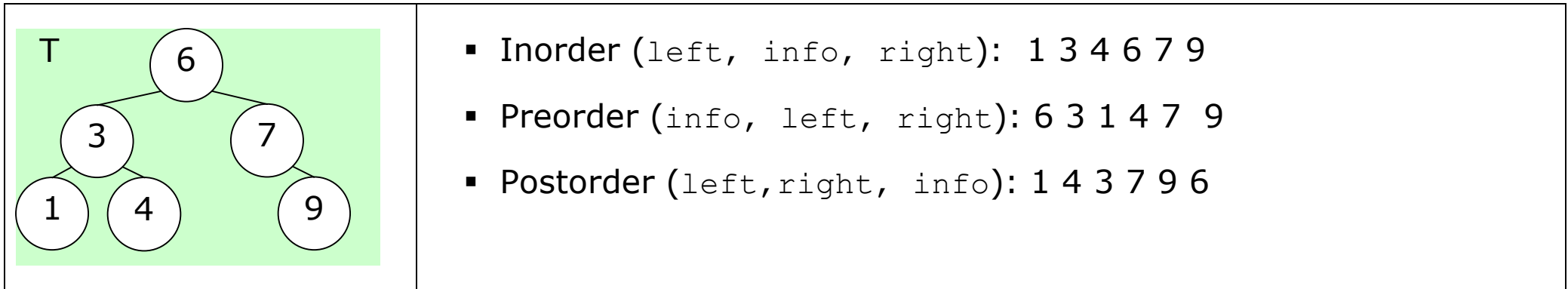
```
$ cat binSearch.cpp
struct tree {                                // tree node
    int info;
    struct tree *left;
    struct tree *right;
};

struct tree *insert(struct tree *p, int info) { // insert new node at leaf,
                                                // return pointer to insert position
    if (p==NULL) {                             // leaf found
        p = new tree;                          // create new node
        p->info = info;                          // store info in the node
        p->left = NULL;
        p->right = NULL;
    } else if (info <= p->info)
        p->left = insert(p->left, info);        // info has to go to left subtree
    else
        p->right = insert(p->right, info);     // info has to go to right subtree

    return p;
}
```

(-> Tafel)

Beim **Durchlaufen** von Binärbäumen können drei Arten unterschieden werden, je nach Reihenfolge der Verwendung der drei Felder des Knotens:



Zum Ausgeben in unserem Beispiel des Suchbaums verwenden wir also inorder, um den Baum sortiert auszugeben:

```
void inorder(struct tree *p) { // print inorder: LDR
    if (p == NULL) return;
    inorder(p->left); // left
    cout << p->info << " "; // info
    inorder(p->right); // right
}
```

Das **Suchen** in einem Binärbaum erfolgt effizient in der Zeit  $O(\log_2(n))$ , da ausgehend von der Wurzel die Suchmenge stets halbiert wird.

Hier bietet es sich ebenfalls an, ein rekursives Programm zu schreiben.

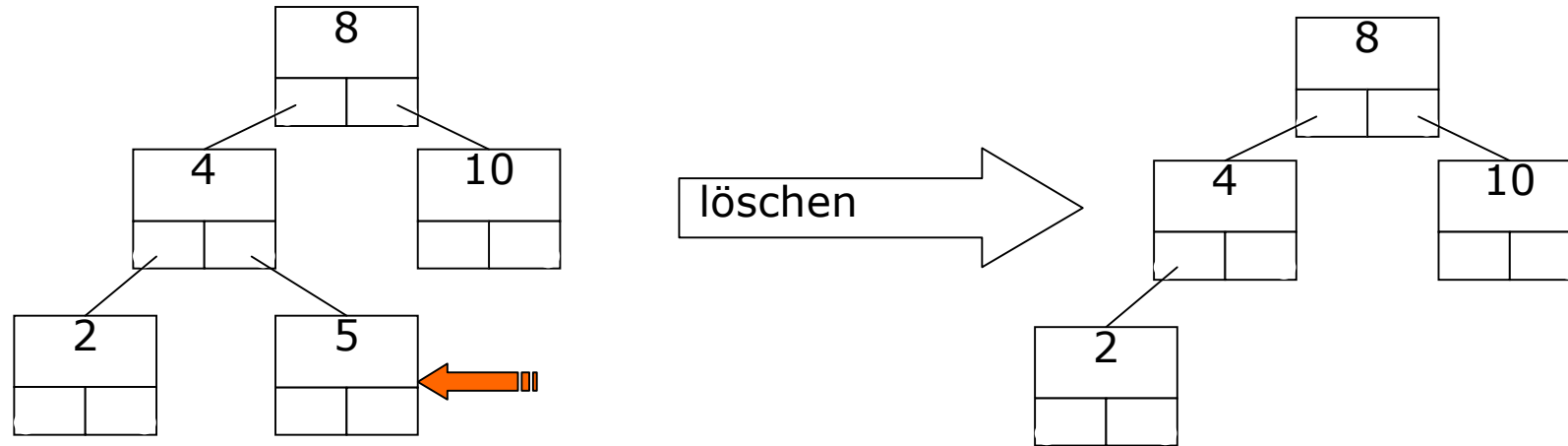
```
bool search(struct tree *p, int info) {           // search info in tree p
    if (p==NULL) return false;
    if (info == p->info) return true;
    else if (info < p->info)
        return search(p->left, info);
    else
        return search(p->right, info);
}
```

Alternativ kann man auch eine Iterative Lösung umsetzen. Nachfolgend eine Funktion, die als Ergebnis einen Zeiger auf einen Knoten liefert, der die Suchinformation enthält, oder NULL ist.

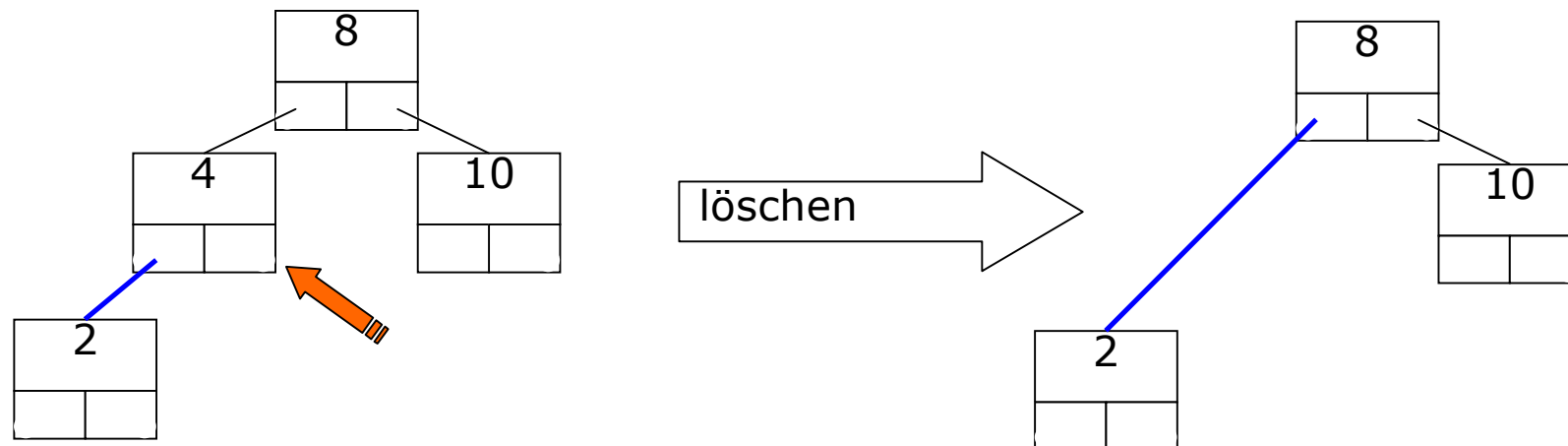
```
struct tree *searchTree(struct tree *p, int key) {
    if (p==NULL) return NULL;           // empty tree
    while (p->info != key) {
        if (key < p->info) p = p->left;
        else p = p->right;
        if (p==NULL) break;
    }
    return p;
}
```

Der schwierigste Part ist das **Löschen** eines Knotens, da hier neu „verzeigert“ werden muss.

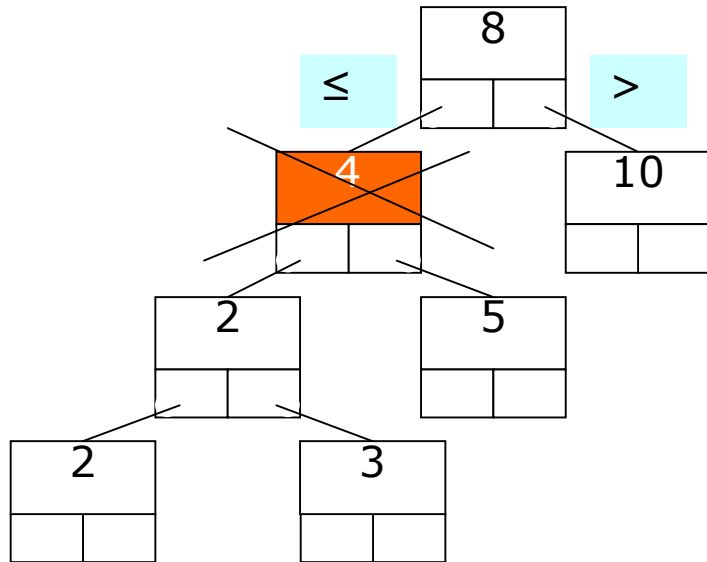
Ein Knoten kann direkt gelöscht werden, wenn er ein **Blatt** ist:



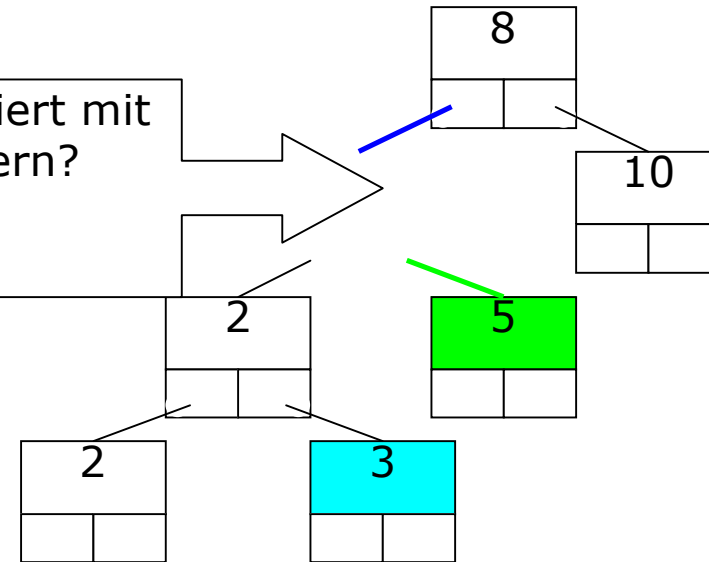
Auch ist das Löschen von **Knoten** mit nur **einem Kind** unproblematisch:



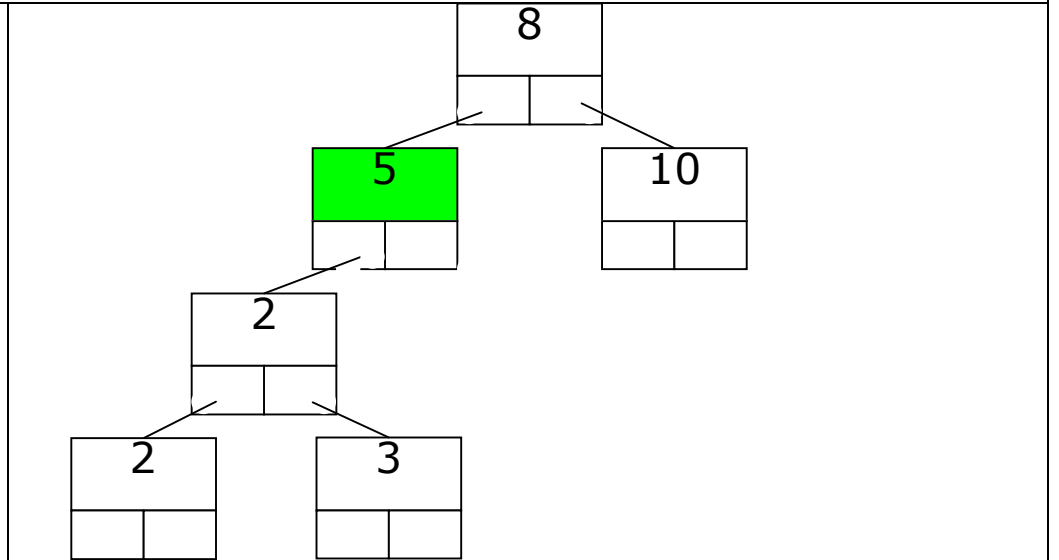
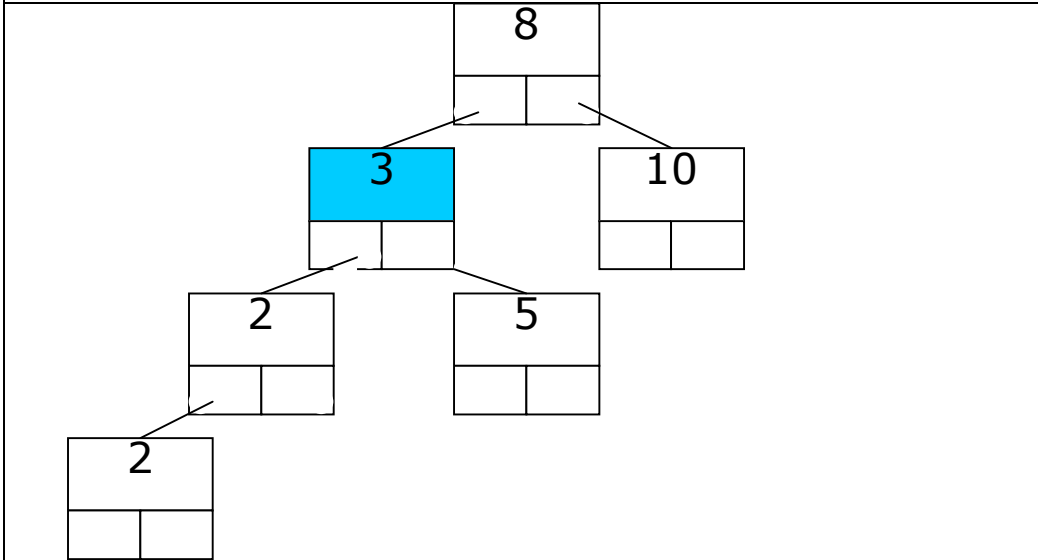
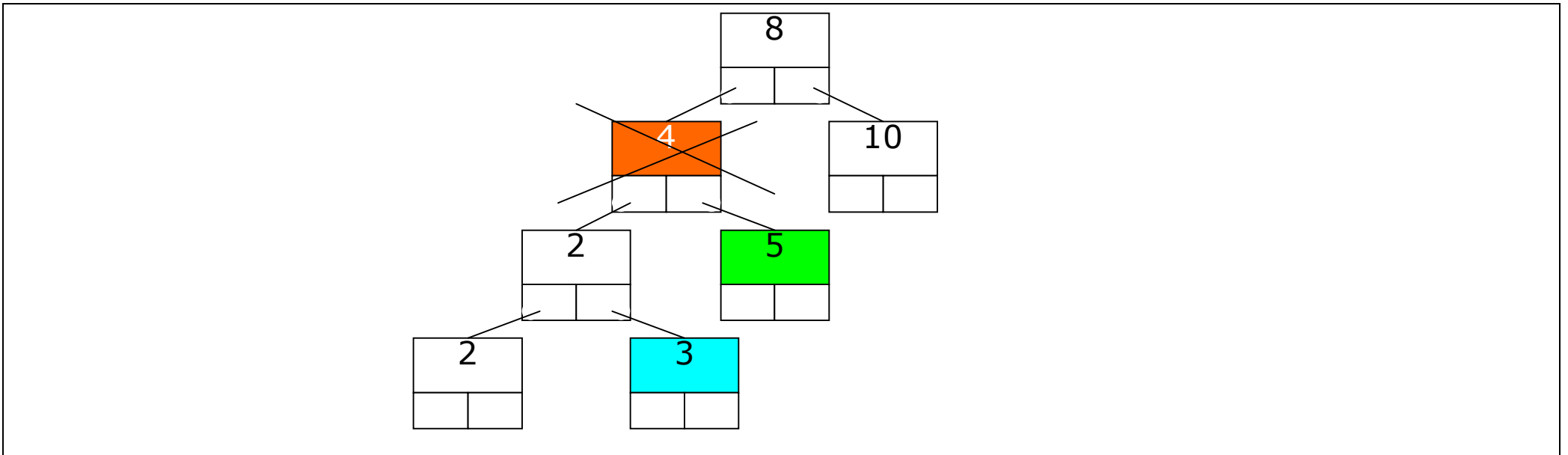
Schwierige ist der Fall, dass ein **Knoten zwei Kinder** hat:



was passiert mit den Zeigern?



Hier gibt es zwei gleichwertige Lösungen: ersetze den Lösch-Knoten durch **größtes Element des linken** oder durch das **kleinste Element des rechten** Unterbaums.



Wir verwenden die **Lösung** mit dem kleinsten Element des rechten Unterbaums und implementieren sie als rekursive Funktion:

```

struct tree *deleteTree(struct tree *p, int key) { // delete node by replacing
                                                    // smallest node of right subtree

    if (key==p->info) { // delete p
        if (p->left==p->right) { // leaf
            delete p; return NULL;
        } else if (p->left==NULL) { // left subtree is empty
            struct tree *p1 = p->right;
            delete p; return p1;
        } else if (p->right==NULL) { // right subtree is empty
            struct tree *p1 = p->left;
            delete p; return p1;
        } else { // both subtrees not empty
            struct tree *p1 = p->right; // for smallest element on right
            struct tree *p2 = p->right;
            while (p1->left != NULL) // search smallest node on right side
                p1 = p1->left;
            p1->left = p->left;
            delete p; return p2;
        }
    }

    if (key < p->info)
        p->left = deleteTree(p->left, key);
    else
        p->right = deleteTree(p->right, key);
    return p;
}

```

Damit sieht das komplette Programm wie folgt aus:

```
$cat binSearch.cpp
#include <iostream.h>
struct tree {                                // node
    int info;
    struct tree *left;
    struct tree *right;
};

struct tree *insert(struct tree *p, int info) { // insert new node at leaf,
                                                // return pointer to insert position
    if (p==NULL) {                             // leaf found
        p = new tree;                          // create new node
        p->info = info;                         // store info in the node
        p->left = NULL;
        p->right = NULL;
    } else if (info <= p->info)
        p->left = insert(p->left, info); // info has to go to left subtree
    else
        p->right = insert(p->right, info); // info has to go to right subtree

    return p;
}

void inorder(struct tree *p) {                 // print inorder: LDR
    if (p == NULL) return;
    inorder(p->left);
    cout << p->info << " ";
    inorder(p->right);
}
```

```

void preorder(struct tree *p) {                                // print preorder DLR
    if (p == NULL) return;
    cout << p->info << " ";
    preorder(p->left);
    preorder(p->right);
}

void postorder(struct tree *p) {                               // print postorder LR=
    if (p == NULL) return;
    postorder(p->left);
    postorder(p->right);
    cout << p->info << " ";
}

bool search(struct tree *p, int key) {                         // search info in tree p
    if (p==NULL) return false;
    if (key == p->info)
        return true;
    else if (key < p->info)
        return search(p->left, key);
    else
        return search(p->right, key);
}

```

```
struct tree *searchTree(struct tree *p, int key) {  
    if (p==NULL)  
        return NULL;                                // empty tree  
    while (p->info != key) {  
        if (key <p->info)  
            p = p->left;  
        else  
            p = p->right;  
        if (p==NULL)  
            break;  
    }  
    return p;  
}
```

```

struct tree *deleteTree(struct tree *p, int key) {// delete node by repllacing
                                                    // smallest node of right subtree

    if (key==p->info) {                               // delete p
        if (p->left==p->right) {                       // leaf
            delete p;
            return NULL;
        } else if (p->left==NULL) {                   // left subtree is empty
            struct tree *p1 = p->right;
            delete p;
            return p1;
        } else if (p->right==NULL) {                  // right subtree is empty
            struct tree *p1 = p->left;
            delete p;
            return p1;
        } else {                                     // both subtrees not empty
            struct tree *p1 = p->right; // for smallest element on right
            struct tree *p2 = p->right;
            while (p1->left != NULL) // search smallest node on right side
                p1 = p1->left;
            p1->left = p->left;
            delete p;
            return p2;
        }
    }

    if (key < p->info)
        p->left = deleteTree(p->left, key);
    else
        p->right = deleteTree(p->right, key);
    return p;
}

```

```

main() {
    struct tree *root=NULL;           // root of tree
    int val;
    while (true) {
        cout << "Integer (exit with neg value): ";
        cin >> val;
        if (val<0) break;
        root = insert(root, val);
    }

    cout << "Inorder: ";
    inorder(root);
    cout << endl;

    cout << "Preorder: ";
    preorder(root);
    cout << endl;
    cout << "Postorder: ";
    postorder(root);
    cout << endl;

    cout << "Serach value: ";
    cin >> val;
    if (search(root, val)) cout << "found!" << endl;
    else cout << "NOT found!" << endl;
    cout << searchTree(root, val) << endl;

    cout << "Delete value: ";
    cin >> val;
    root = deleteTree(root, val);
    cout << "Inorder: ";
    inorder(root);
    cout << endl;
}
$

```

## 1.3. Operationen auf Binärbäumen

In größeren Anwendungen sind Operationen auf ganzen Bäumen erforderlich, wie Kopieren oder der Test auf Gleichheit.

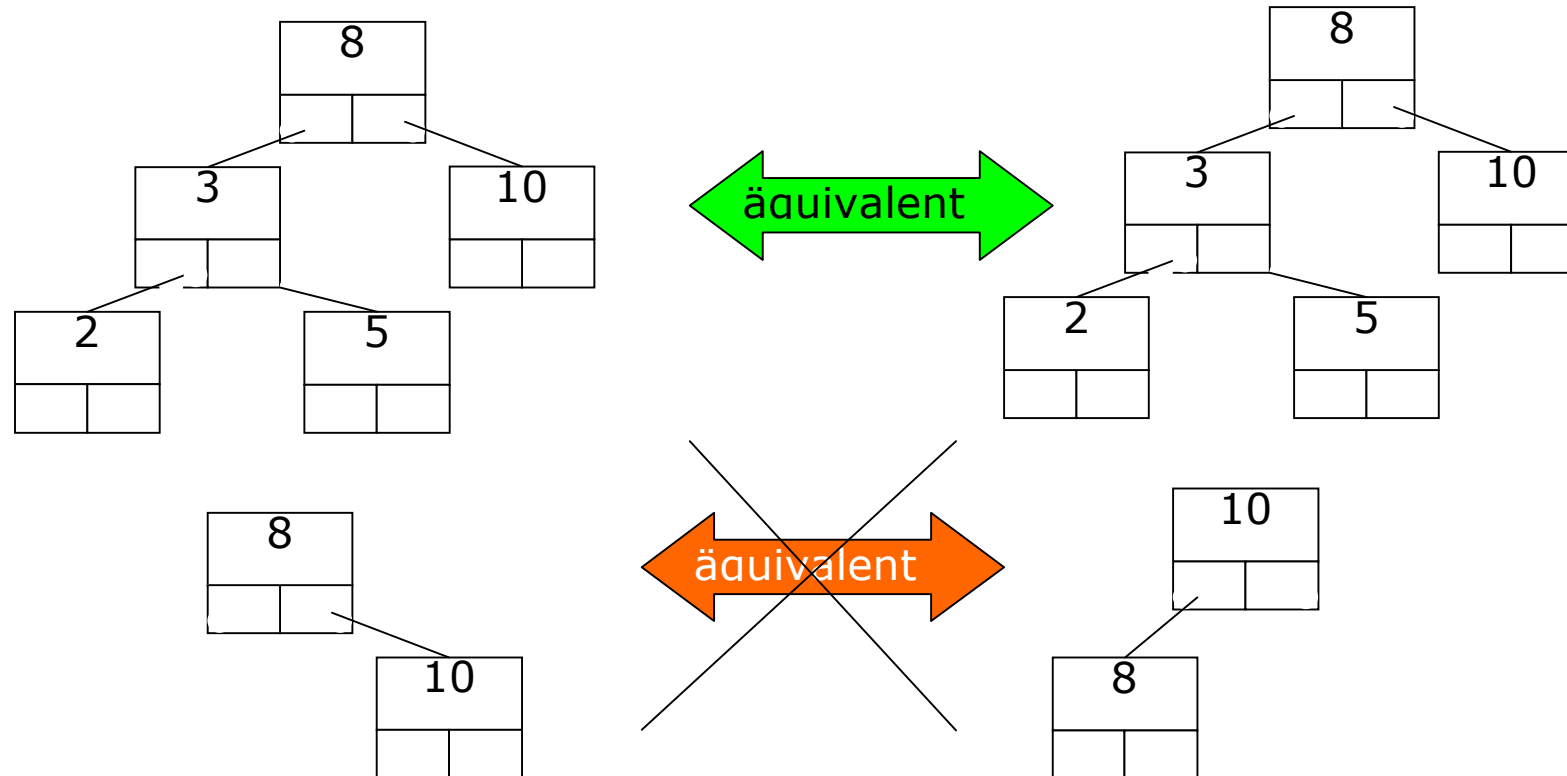
### 1.3.1. Kopieren von Bäumen

Auch hier bietet es sich an, die rekursive Struktur mittels einer rekursiven Funktion zu Durchlaufen und jeweils Knotenkopien zu erzeugen.

```
struct tree *copy(struct tree *t) { // returns a pointer to an exact copy of t
    if (t == NULL) return NULL;
    struct tree *l, *r, *n;
    l = copy(t->left); // copy left subtree
    r = copy(t->right); // copy right subtree
    n = new tree; // create new node
    n->info = t->info; // store info into the node
    n->left = l; // let left subtree become l
    n->right = r; // let left subtree become r
    return n;
}
```

### 1.3.2. Äquivalenz von Bäumen

Zwei Bäume sind äquivalent, wenn sie die gleiche topologische Struktur aufweisen und an korrespondierenden Stellen den gleichen Wert haben.



#### Hörsaalübung:

Entwickeln Sie eine Funktion

```
bool isEqual(struct tree *t1, struct tree *t2)
```

die dann `true` liefert, wenn die Bäume `t1` und `t2` äquivalent sind.

## 1.4.Repräsentation von Bäumen durch Binärbäume

Die Darstellung von k-nären Bäumen, bei denen jeder Knoten unterschiedlichen Grad haben kann, könnte erfolgen, indem der Maximalgrad k festgelegt wird, und eine Struktur definiert wird der

Form:

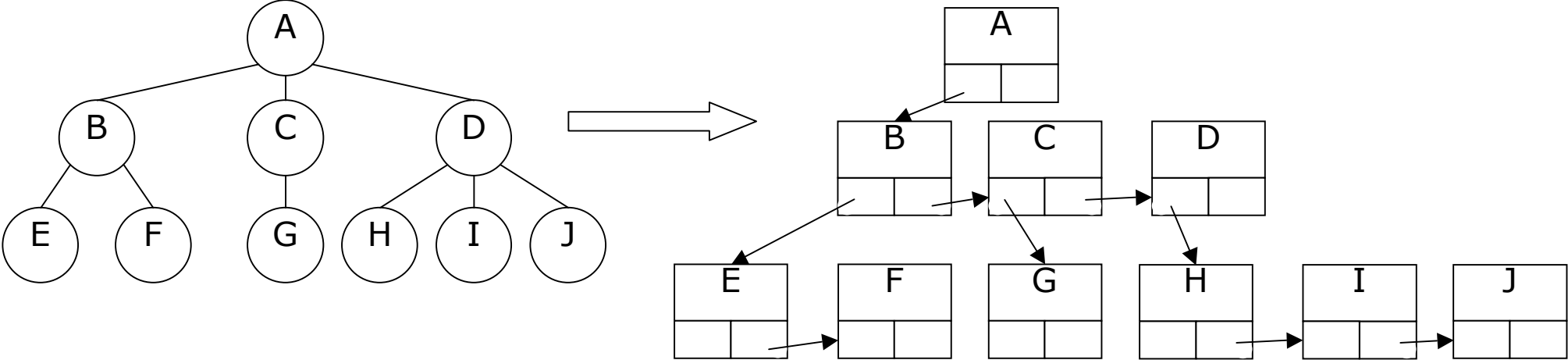
info			
Kind 1	Kind 2	...	Kind k

Der Nachteil dieser Lösung ist, dass man wegen der konstanten Anzahl von Kindern Platz verschwendet, wenn nur einige Knoten k Kinder haben, der Rest aber viel weniger. Weiterhin ist dies umständlich zu Programmieren.

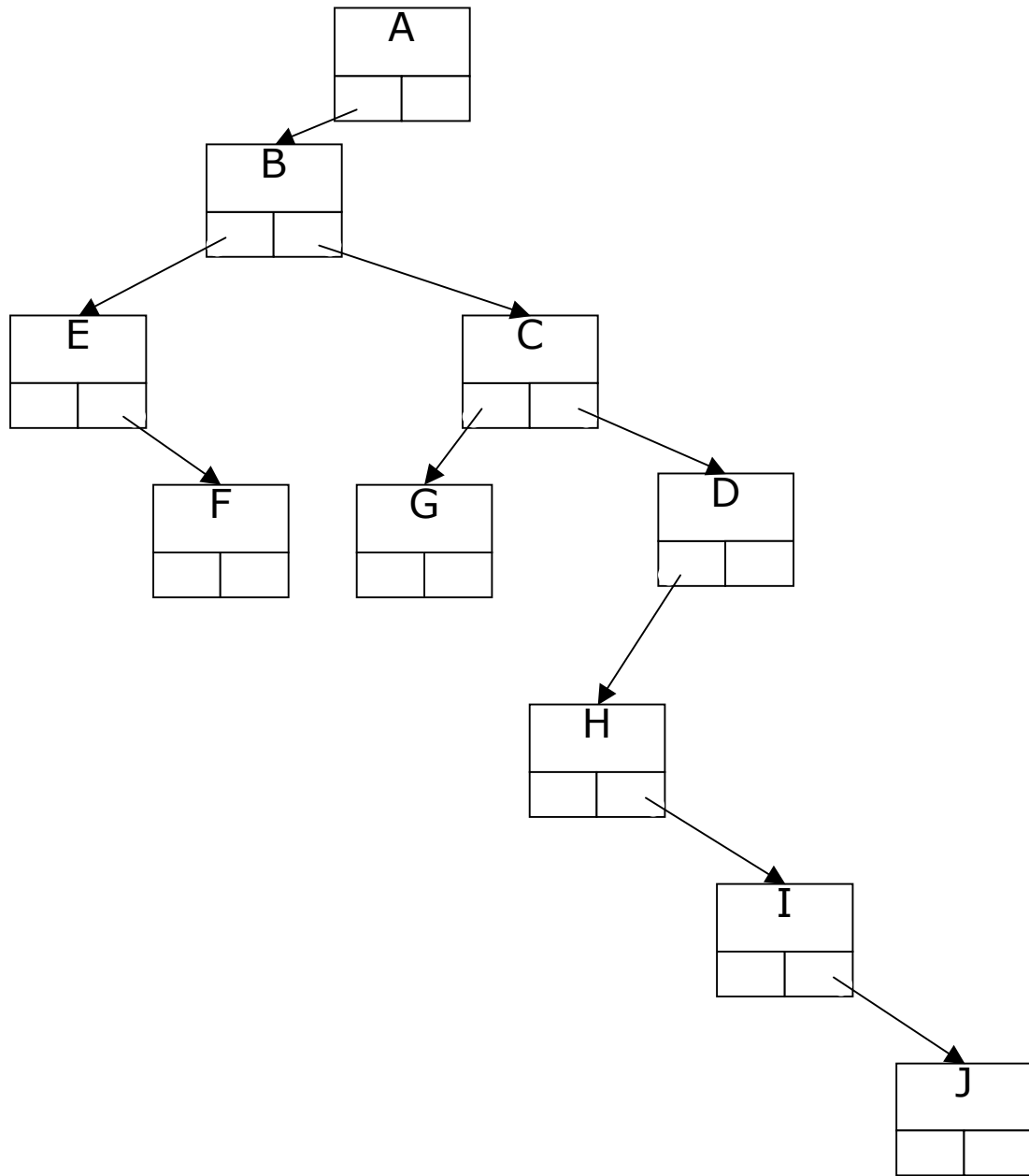
Eine elegante Methode ist es, einen k-nären Baum durch einen Binärbaum darzustellen, bei dem ein Knoten folgende Struktur aufweist:

info	
Kind	Geschwister

Beispiel:



Wenn man die Knoten etwas anders arrangiert, erkennt man den Binärbaum besser:



## Hörsaalübung

Entwickeln Sie einen ADT CBinaryTree für binäre Bäume. Die Implementierung soll auf den o.a. Funktionen mittels Zeiger beruhen.