

# Knotenbasierte Datenstruktur: Baum

Vorgiff: Graphen verbinden generell informationstragende Knoten einer Knotenmenge über Kanten miteinander (nicht unbedingt alle Kanten)

Spezialfall der Graphen: Bäume

- = zyklensfreie Graphen (jeder Knoten hat **nur genau 1 Vorgänger**, zwischen der Wurzel und jedem Knoten gibt es genau einen Pfad)
- alle Knoten eines Baumes sind auf irgendeinem Weg miteinander verbunden
- einer der Knoten wird als Wurzelknoten festgelegt, und als hierarchisch höchster Knoten dargestellt (da wir von oben nach unten lesen)

Bäume sind nichtlineare Speicherstrukturen, ihre Knoten sind im Speicher verteilt

Sie verallgemeinern das Listenprinzip:

- ein Knoten kann **mehr als einen Nachfolger** haben
- Knoten ohne Nachfolger bezeichnen wir als *Blätter*
- der Wurzelknoten ist der einzige Knoten ohne Vorgänger (darauf zeigt der Anker)

**Auch Bäume speichern wie Listen und Arrays nur Datenobjekte des gleichen Typs.**

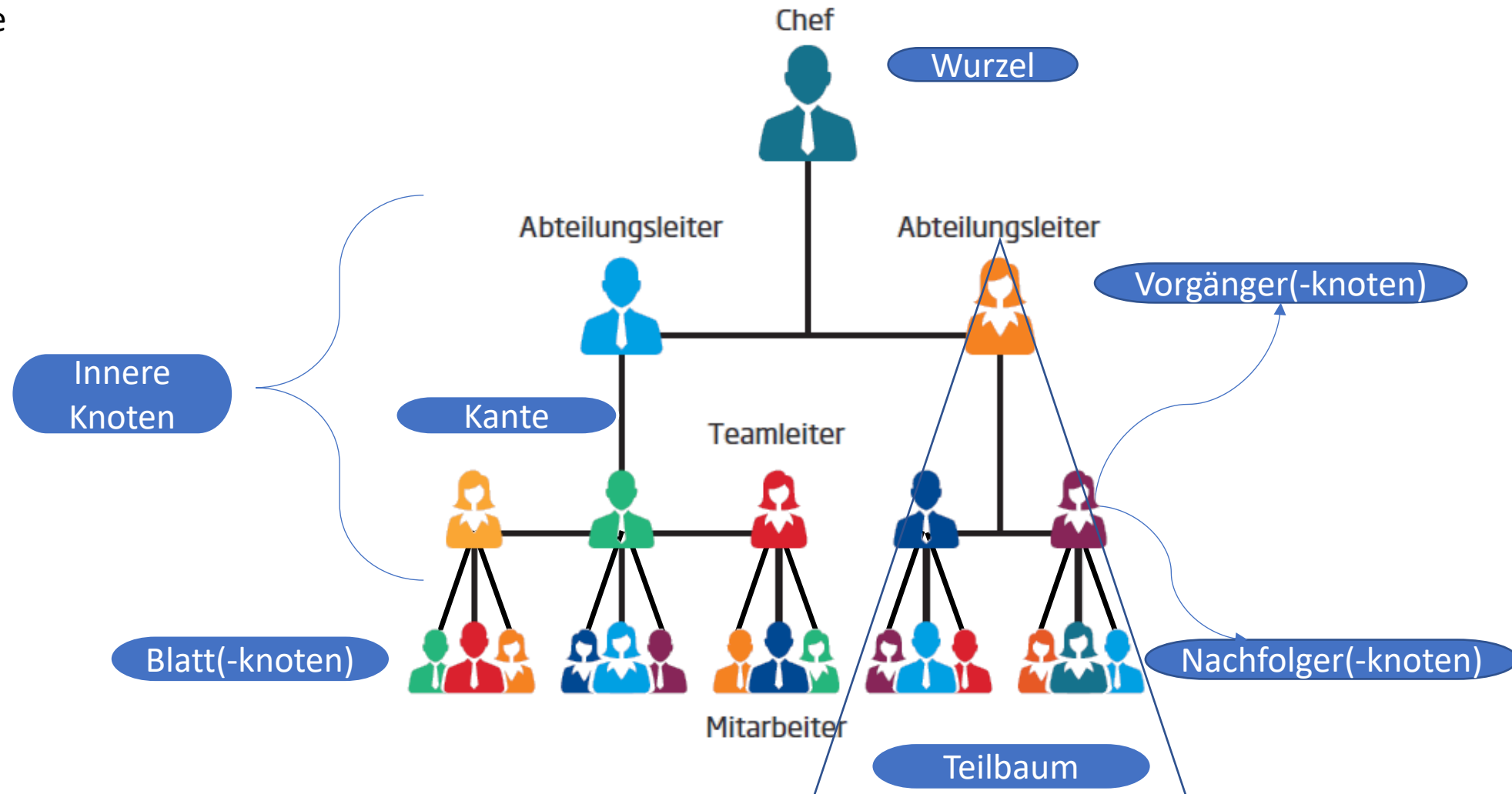
# Bäume: Begriffe

Ebene  
0

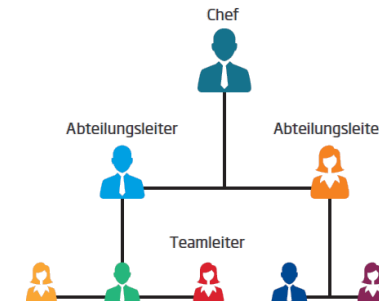
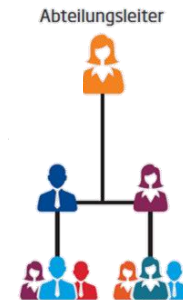
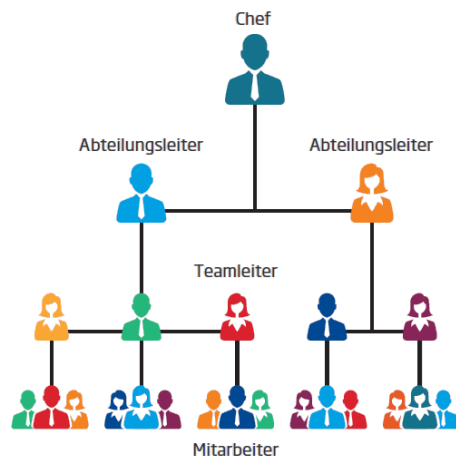
1

2

3



# Bäume: weitere Begriffe



Mehrere Bäume mit jeweils eigenen Wurzeln nennen wir Wald.

# Binäre Suchbäume

**Sortierte Arrays:** können mit binärer Suche jeden Wert in  $O(\log N)$  – Geschwindigkeit finden.

Problem sortierter Arrays:

Sind bei Einfügungen und Löschungen langsam ( $N$  Schritte im worst case (1. Wert), und  $N / 2$  im Average case).

**Hash-Tabellen:** haben  $O(1)$ -Geschwindigkeit für Zugriff, Einfügen und Löschen

Nachteil von Hash-Tabellen: sie bewahren keine Sortierung.

Wir suchen eine Datenstruktur, die die Sortierung bewahrt, und schnelle Suchen, Einfügen und Löschen unterstützt!  
Weder das sortierte Array noch die Hashtabelle sind dafür ideal...

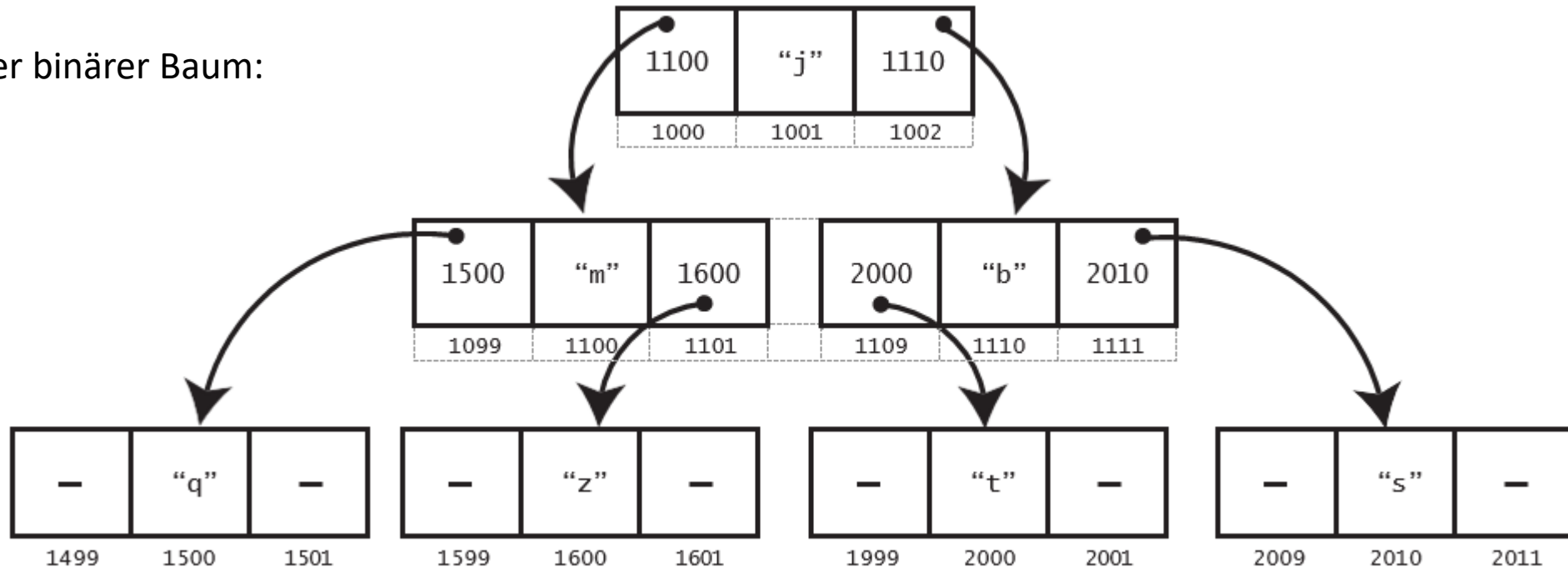
**Willkommen zu den Binären Suchbäumen!**

# Binäre Bäume

Ein *Baum* ist eine knotenbasierte Datenstruktur, aber in einem Baum kann jeder Knoten (im Gegensatz zur Liste) *mehrere* Nachfolgerknoten haben.

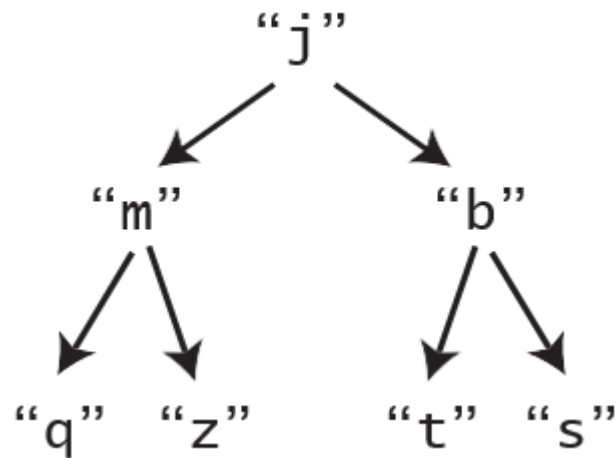
Im *binären* Baum hat jeder Knoten Verweise auf bis zu 2 Nachfolgerknoten.

Ein einfacher binärer Baum:



# Binäre Bäume

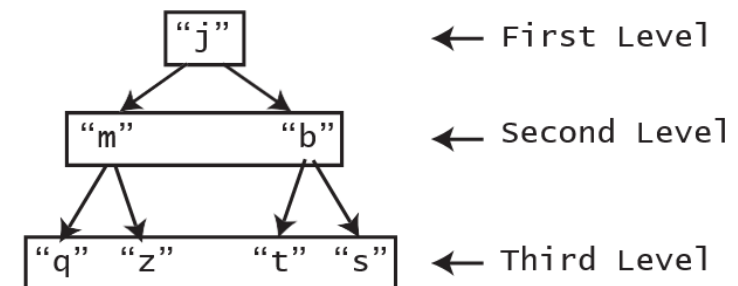
Ohne Speicheradressen:



Bäume haben eine eigene Nomenklatur:

- *Wurzel*: oberster Knoten (im Beispiel: "j")
- *Elternknoten*: "j" ist der *Elternknoten* von "m" und "b"
- *Kindknoten*: "m" und "b" sind *Kindknoten* von "j"
- *Elternknoten*: "m" ist *Elternknoten* von "q" und "z" (*Kindknoten* von "m")
- *Nachfahren/ Nachfolger*: *alle* Knoten, die von ihm abstammen (alle Knoten sind Nachfahren/ Nachfolger von "j")
- *Vorfahren*: *alle* Knoten, von denen ein Knoten abstammt (j": Vorfahre aller Knoten des Baumes)

- Bäume haben Ebenen.  
Jede Ebene ist eine Zeile im Baum.  
Unser Beispiel hat 3 Ebenen:



# Binäre **Such**bäume (BST: Binary Search Tree)

Unter vielen Arten von Bäumen fokussieren wir uns auf den Binären Suchbaum:

Zwei wichtige Adjektive: *binär* and *Such*-.

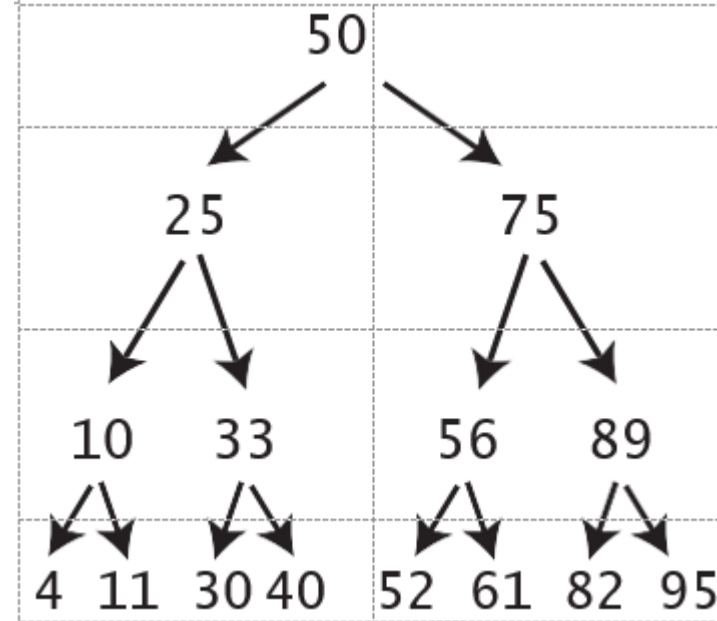
In einem **binären** Baum hat jeder Knoten 0-2 Kindknoten.

Ein **binärer Suchbaum** ist ein binärer Baum, für den folgende Sortier-Regel gilt:

- Die Werte der linken Nachfahren jedes Knotens (gesamter Teilbaum) müssen kleiner (oder gleich) dem Wert des Knotens selbst sein
- Die Werte der rechten Nachfahren (gesamter Teilbaum) müssen größer (oder gleich) dem des Knotens selbst sein..

# Binäre Suchbäume

Binärer Suchbaum (Werte der Knoten sind hier Zahlen):



Jeder Knoten hat (optional) einen **linken Kindknoten (Teilbaum)** mit einem kleineren Wert/ Werten als er selbst, und (optional) einen **rechten Kindknoten (Teilbaum)** mit einem Wert/ Werten größer seinem eigenen.

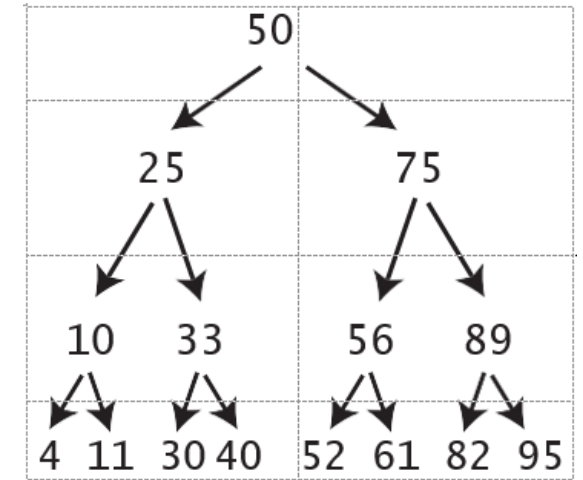
Diese Struktur des binären Suchbaums erlaubt eine sehr, sehr schnelle Suche nach Werten!



# Suche in Binären Suchbäumen

Der Algorithmus der Suche nach einem Wert beginnt in binären Suchbäumen an der Wurzel:

1. Wir prüfen den Wert des Knotens.
2. Ist er der Gesuchte: fertig!
3. Ist der gesuchte Wert kleiner als der aktuelle, suchen wir im linken Teilbaum (nächster Knoten) weiter (#1).
4. Ist der gesuchte Wert größer als der aktuelle, suchen wir im rechten Teilbaum (nächster Knoten) weiter (#1).

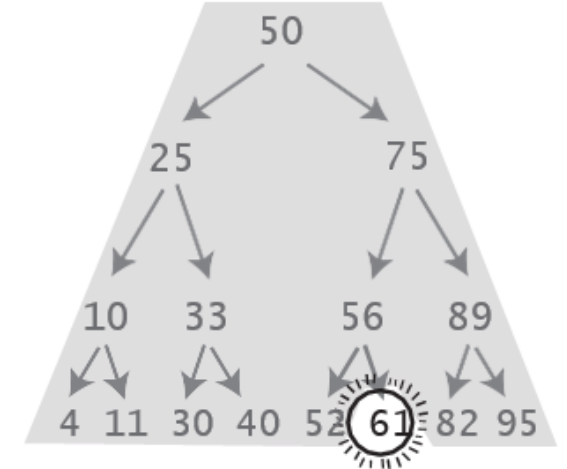
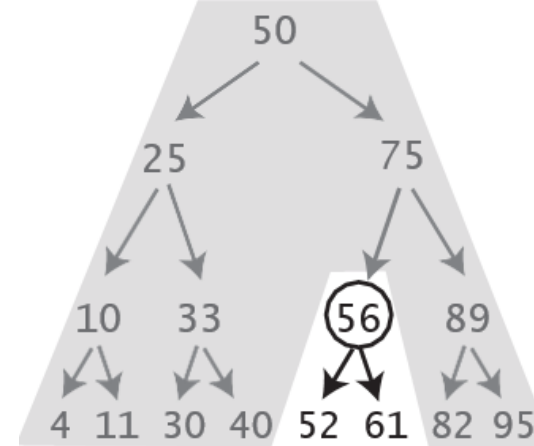
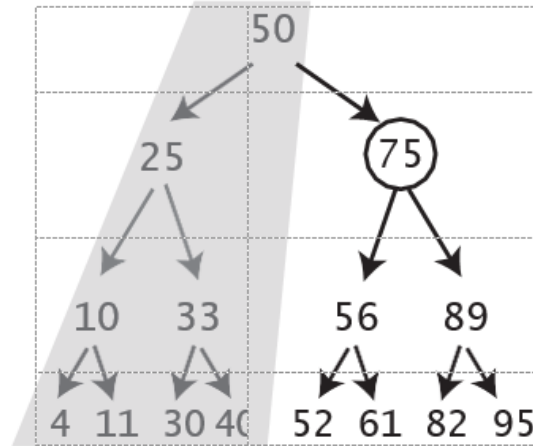
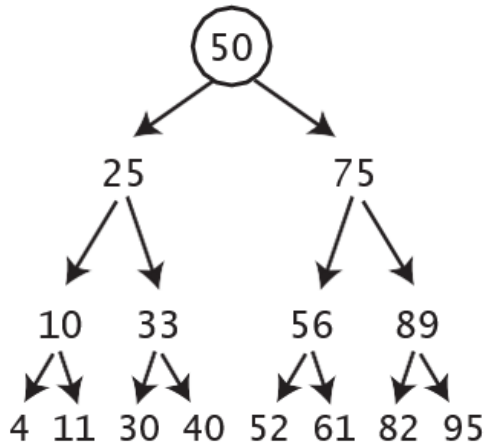


Wir suchen im Beispiel nach dem Wert 61:

Wieviele Schritte brauchen wir für das komplette Durchsuchen?

Wir beginnen bei der Suche im Baum immer an der Wurzel:

# Suche in Binären Suchbäumen



Vier vergleichende Schritte, um den Wert zu finden:

- jeder Schritt entfernt die Hälfte der restlichen Knoten aus der Suche
- die Suche in einem binären Baum kostet  $O(\log N)$ , (in einem perfekt balancierten/ ausgeglichenen binären Suchbaum, dem best-case-Szenario)

*Hat ein balancierter binärer Suchbaum  $N$  Knoten, dann hat er  $\log N$  Ebenen!*

# Suche in Binären Suchbäumen

Warum ist das so?

Angenommen, jede Zeile des Baumes ist komplett mit Knoten gefüllt, ohne freie Stellen:

Dann hat jede neu hinzugefügte komplette Zeile doppelt so viele Knoten wie die vorige (da jeder Knoten 2 Kinder in der nächsten Ebene hat).

Das ist das Muster von  $\log(N)$ , da jede neue Zeile die Daten fast verdoppelt.

Die Tabelle zeigt, wieviele Knoten Bäume in Abhängigkeit ihrer Ebenenzahl haben:

N Nodes	N Rows	$\log N$ (approx)
1	1	0
3	2	2
7	3	3
15	4	4
31	5	5
63	6	6
127	7	7
255	8	8

# Suche in Binären Suchbäumen

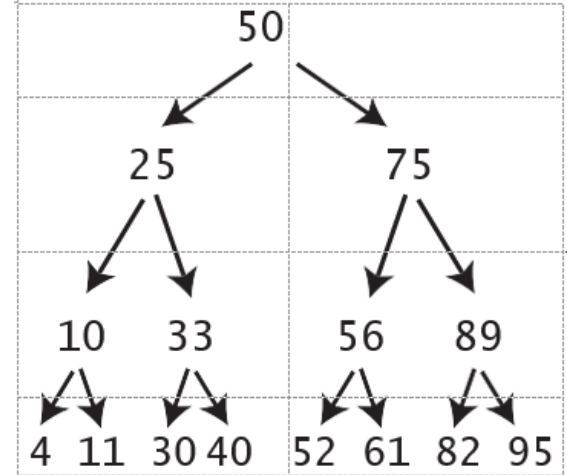
Jeder Suchschritt bewegt uns auf die nächsttiefere Ebene – wir machen maximal so viele Schritte, wie der Baum Ebenen hat.

Deshalb kostet die Suche  **$O(\log N)$** .

Die **Suche** in einem binären Suchbaum hat die **gleiche Effizienz** wie die **binäre Suche in einem sortierten Array**.

Binäre Suchbäume übertrumpfen jedoch die sortierten Arrays beim **Einfügen!**

# Suche in Binären Suchbäumen



Die Implementierung der Such- (und anderen) Operation benutzt intensiv Rekursionen.

Grund:

Rekursion ist das Mittel der Wahl bei Datenstrukturen mit unbekannter Tiefe der Ebenen (vgl. Filesystem).

```
struct treeNode {struct treeNode *pLeft; int data; struct treeNode *pRight;};
```

```
struct treeNode* search (int searchValue, struct treeNode* currentNode)
{
    // Exit: If the node is nonexistent or we've found the value we're looking for:
    if ((NULL == currentNode) || (currentNode->data == searchValue)) return currentNode;
    else
    {
        // If the value is less than the current node, perform search on the left child:
        if (searchValue < currentNode->data) return search(searchValue, currentNode->pLeft);
        // If the value is more than the current node, perform search on the right child:
        else return search(searchValue, currentNode->pRight); // searchValue > node.value
    }
}
```

# Suche in Binären Suchbäumen

```

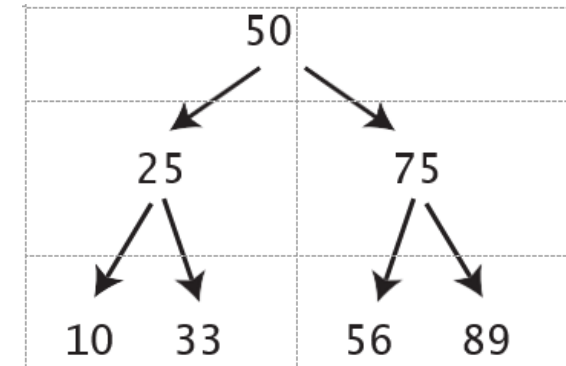
struct treeNode * CreateNode (void)
{
    struct treeNode *pTemp=malloc(sizeof(struct treeNode));
    pTemp->pLeft=NULL;
    pTemp->pRight=NULL;
    return (pTemp);
}

int main(int argc, char** argv)
{
    struct treeNode *pRoot=malloc(sizeof(struct treeNode));
    if (pRoot)
    {
        pRoot->data=50;
        pRoot->pLeft=CreateNode();
        pRoot->pLeft->data=25;
        pRoot->pRight=CreateNode();
        pRoot->pRight->data=75;
        pRoot->pLeft->pLeft=CreateNode();
        pRoot->pLeft->pLeft->data=10;
        pRoot->pLeft->pRight=CreateNode();
        pRoot->pLeft->pRight->data=33;

        pRoot->pRight->pLeft=CreateNode();
        pRoot->pRight->pLeft->data=56;
        pRoot->pRight->pLeft->pLeft=NULL;
        pRoot->pRight->pLeft->pRight=NULL;
        pRoot->pRight->pRight=CreateNode();
        pRoot->pRight->pRight->data=89;
        pRoot->pRight->pRight->pLeft=NULL;
        pRoot->pRight->pRight->pRight=NULL;

        printf ("33 =%d\n", (search (33, pRoot))->data);
    }
    return (EXIT_SUCCESS);
}

```

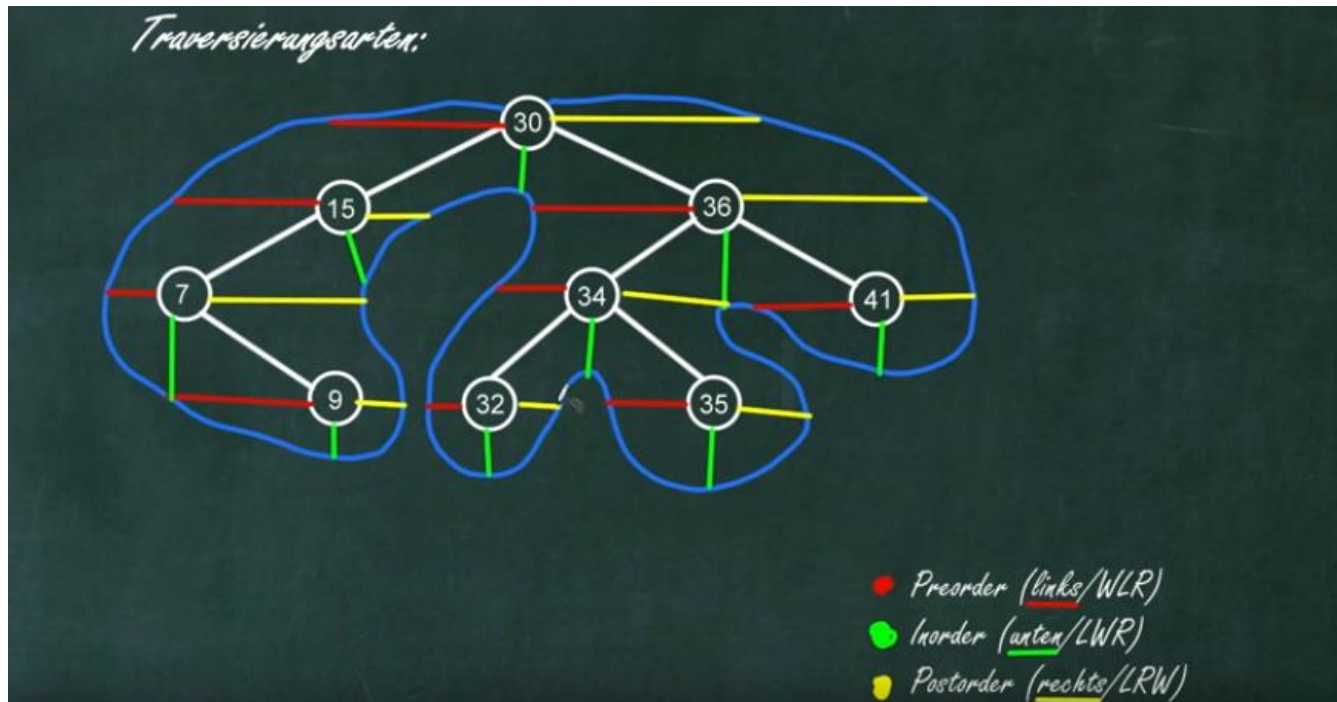


# Suche in Binären Suchbäumen

Varianten der Traversierung binärer Bäume:

Anwendungen: Listenausgabe der Knoten, Suche, ...

[https://www.youtube.com/watch?v=5X8CkFBq\\_8k](https://www.youtube.com/watch?v=5X8CkFBq_8k)



**Level order:** Eine Ebene (level) nach der anderen

**Preorder (WLR):** erst Wurzel/ Knoten, dann links, dann rechts  
(„Pre“: Wurzel zuerst)

**Inorder (LWR):** erst links, dann Wurzel/ Knoten, dann rechts  
(„In“: Wurzel in der Mitte)

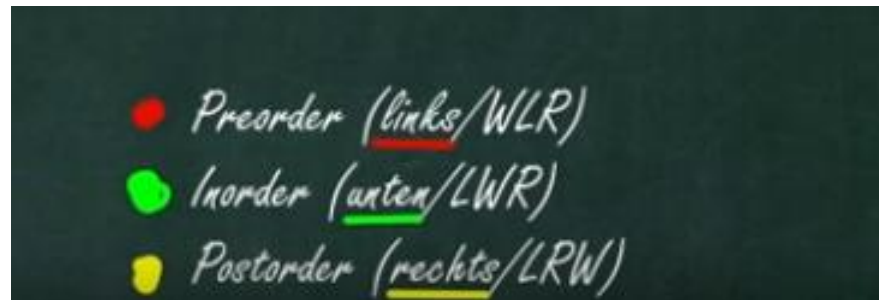
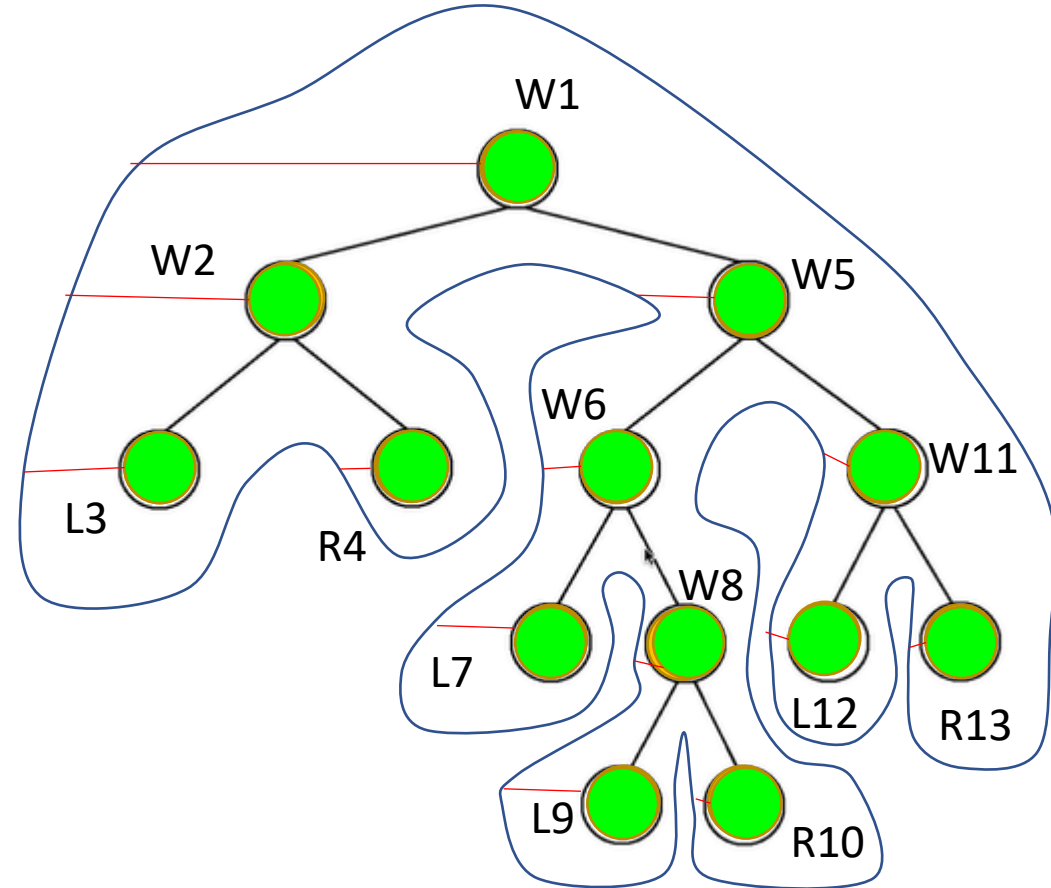
**Postorder (LRW):** erst links, dann rechts, dann Wurzel/ Knoten  
(„Post“: Wurzel danach)

**Regel bei jedem Knoten prüfen und einhalten!**

# Suche in Binären Suchbäumen : Preorder

Tiefensuche

W – L - R auf jeden Knoten anwenden!



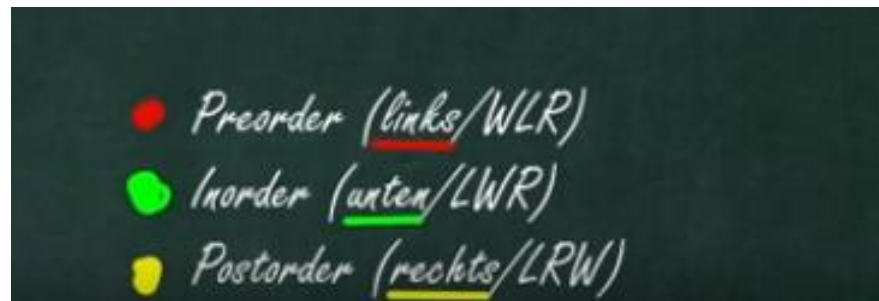
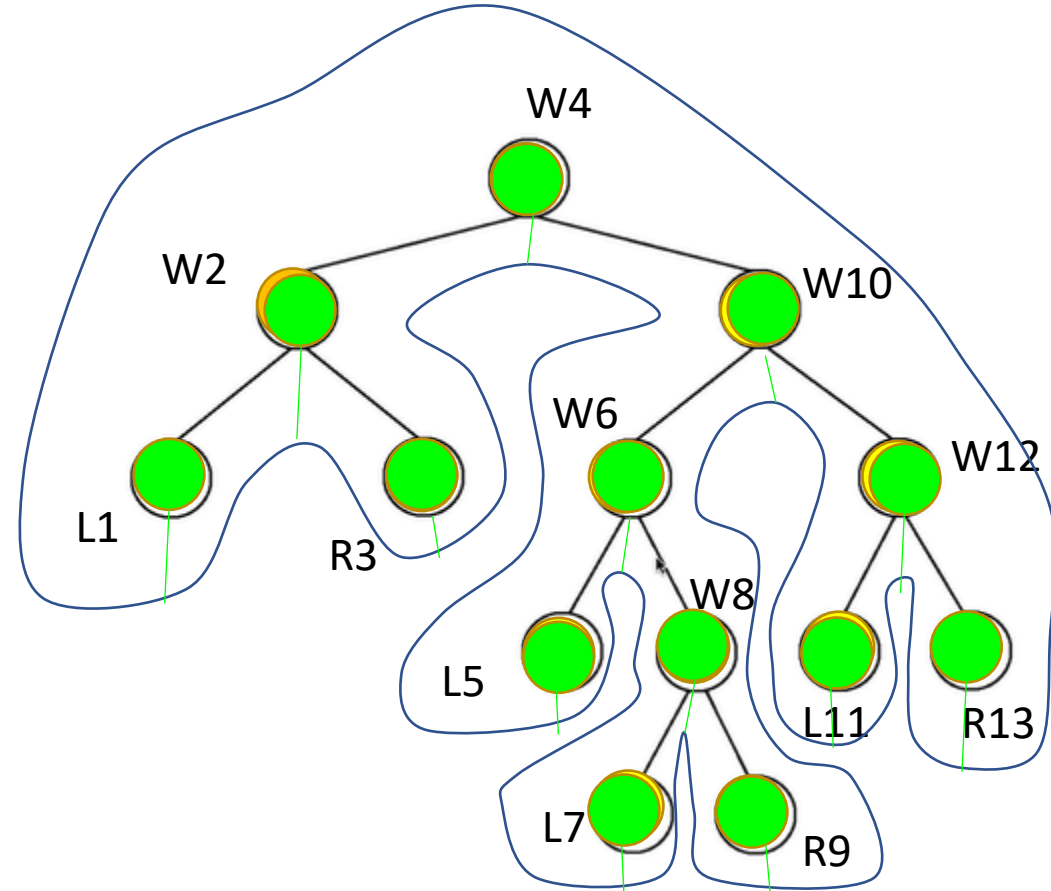


# Suche in Binären Suchbäumen: Inorder

Reihenfolge in binären **Such**bäumen,

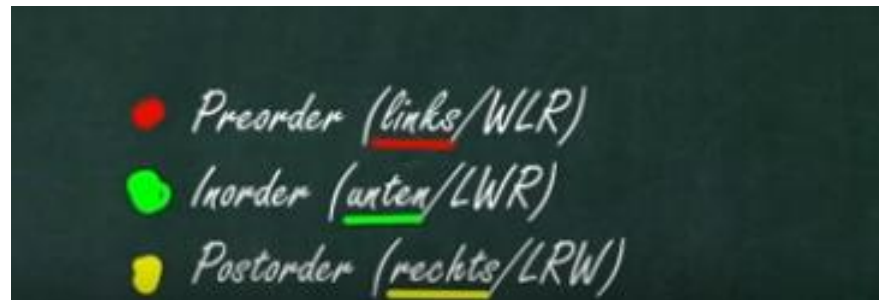
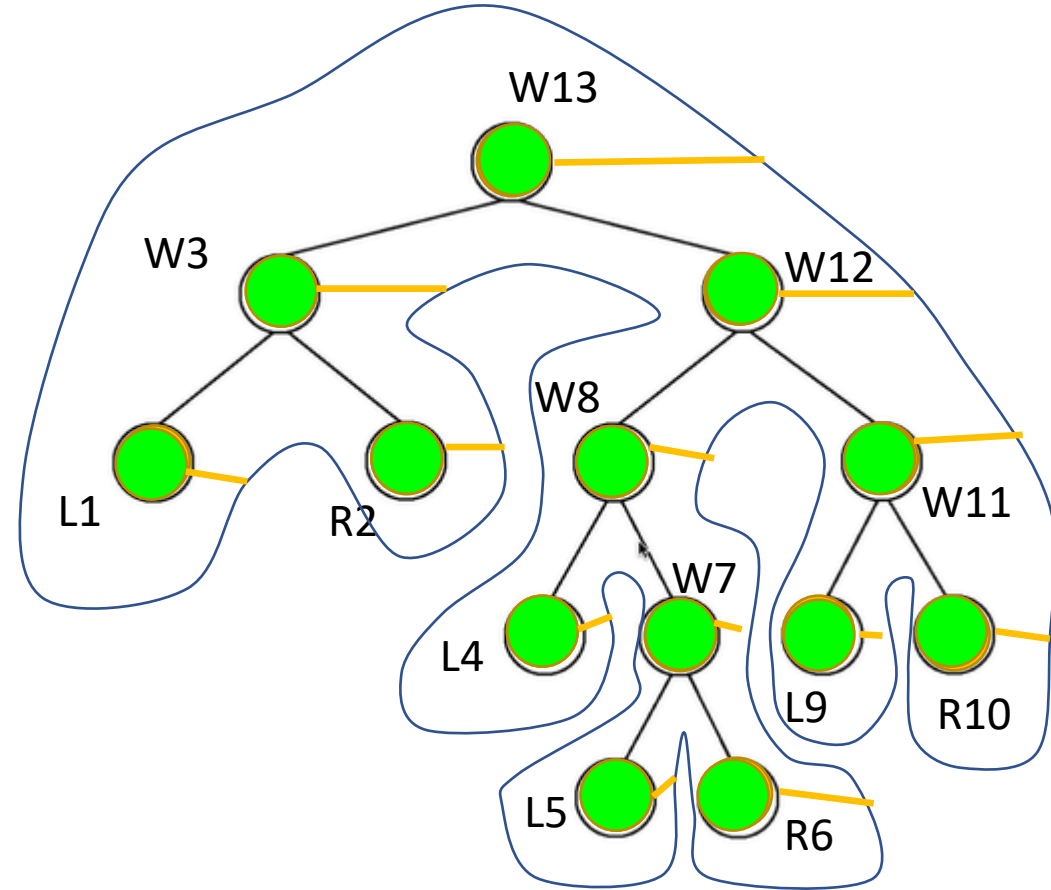
Hier:  $x - y/z + a/c + 7*b$

L – W – R auf jeden Knoten anwenden!



# Suche in Binären Suchbäumen: Postorder

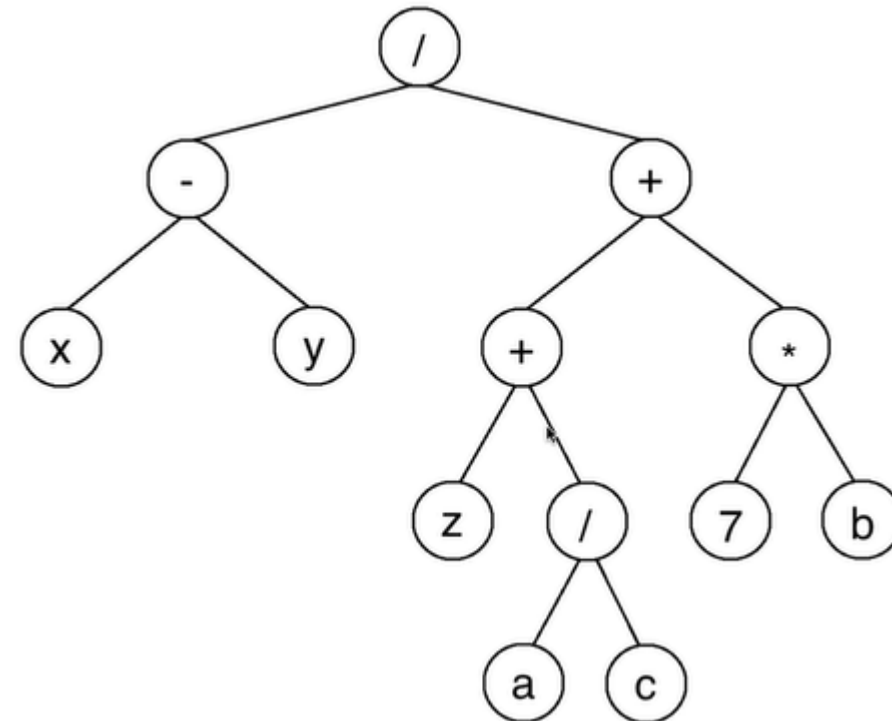
L – R – W auf jeden Knoten anwenden!



# Suche in Binären Suchbäumen: Level order

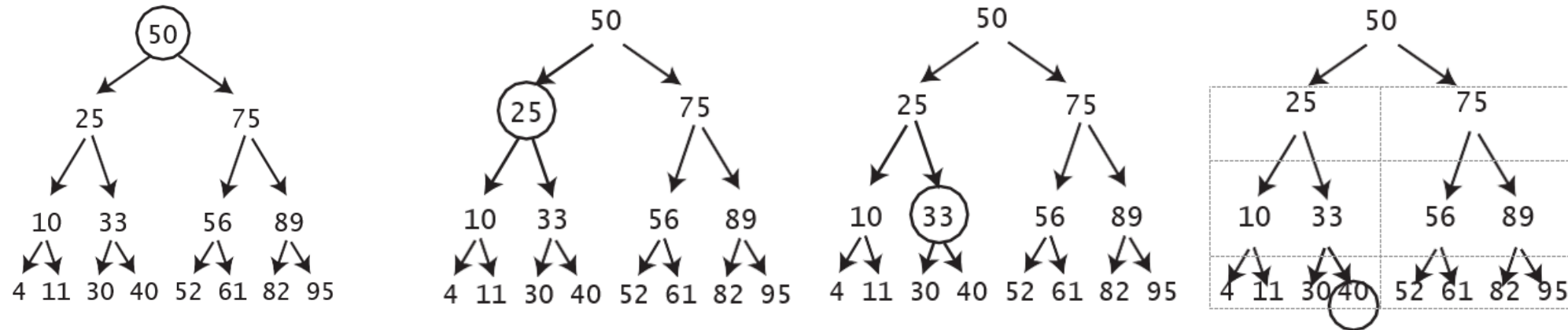
Weitere Traversierungen: Breitensuche

Level-order: / - + x y + \* z / 7 b a c



# Einfügen in Binären Suchbäumen (o. Duplikate)

Lassen Sie uns den Wert 45 in unseren Beispielbaum einfügen: (an einen Knoten mit  $< 2$  Kindknoten!)



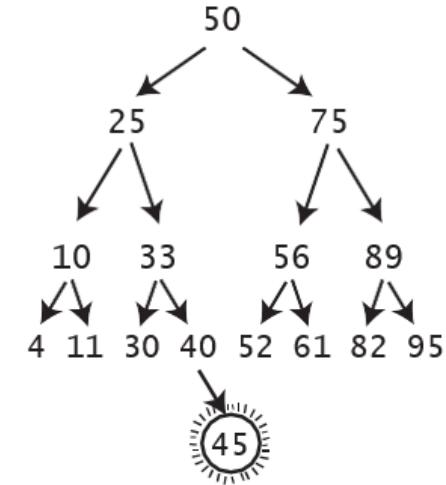
Wir haben jetzt einen Knoten ohne (also  $< 2$ ) Kinder erreicht, deshalb müssen wir nicht mehr weitermachen.

Wir können den Knoten jetzt einfügen.

# Einfügen in Binären Suchbäumen (o. Duplikate)

Da 45 größer als 40 ist, fügen wir ihn als rechten Kindknoten von 40 ein:

Das Einfügen brauchte 5 Schritte, davon 4 Suchschritte, und einen Einfügungsschritt.



Einfügen in einen binären Suchbaum braucht im worst case  $(\log N) + 1$  Schritt:  $O(\log N)$

In ein sortiertes Array hingegen dauert das Einfügen im worst case  $O(N)$  (Verschiebungen).

Binäre Suchbäume haben  $O(\log N)$  Suchen *und*  $O(\log N)$  Einfügungen  
(das ist relevant in Anwendungen, wo man mit vielen Datenänderungen rechnen muss).

# Einfügen: Implementierung (o. Duplikate)

```
struct treeNode* insert (int newValue, struct treeNode* currentNode)
{
    if (newValue < currentNode->data )
    { // If the left child does not exist: insert the value as left child
        if (NULL == currentNode->pLeft)
        {
            struct treeNode *newNode = CreateNode(); // auf NULL testen!
            newNode->data=newValue;
            currentNode->pLeft = newNode;
        }
        else    insert(newValue, currentNode->pLeft);
    }
    else
    {
        if (newValue > currentNode->data)
        { // If the right child does not exist: insert the value as right child
            if (NULL == currentNode->pRight)
            {
                struct treeNode *newNode = CreateNode(); // auf NULL testen!
                newNode->data=newValue;
                currentNode->pRight = newNode;
            }
            else    insert(newValue, currentNode->pRight);
        }
    }
}
```

Aufruf:

```
insert (45, pRoot);
printf ("45 =%d\n", (search (45, pRoot))->data);
```

# Einfügen: Implementierung

Baum kann beim Einfügen unausgeglichen (imbalanced) und weniger effizient werden (versuchen Sie, 1-2-3-4-5 einzufügen – lineare Struktur entsteht!)

-> vor dem Einfügen eines sortierten Arrays in einen binären Suchbaum die Datenreihenfolge lieber randomisieren!

**Worst-case Szenario:** Baum ist komplett unausgeglichen: Suche kostet  $O(N)$  wie in einer sortierten Liste.

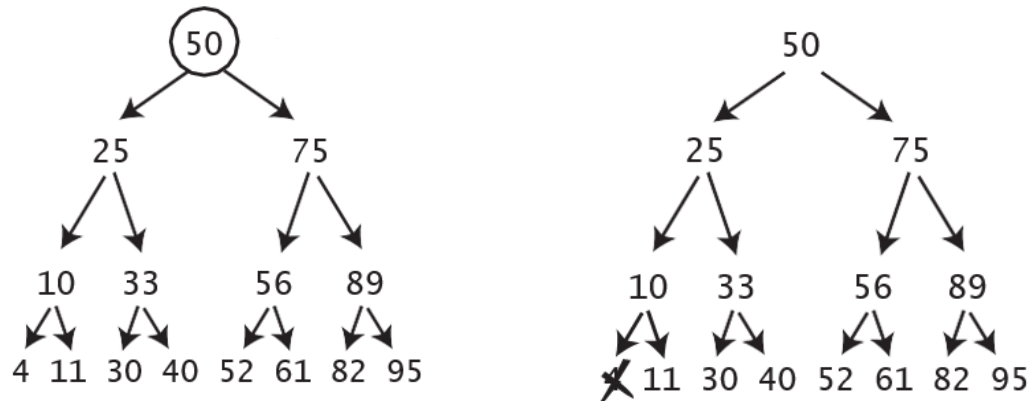
**Best-case Szenario:** perfekt ausgeglichener Baum: Suche kostet  $O(\log N)$ .

Typischerweise werden Daten in zufälliger Reihenfolge einsortiert, ein Baum ist ziemlich ausgeglichen und die Suche kostet ca.  $O(\log N)$ .

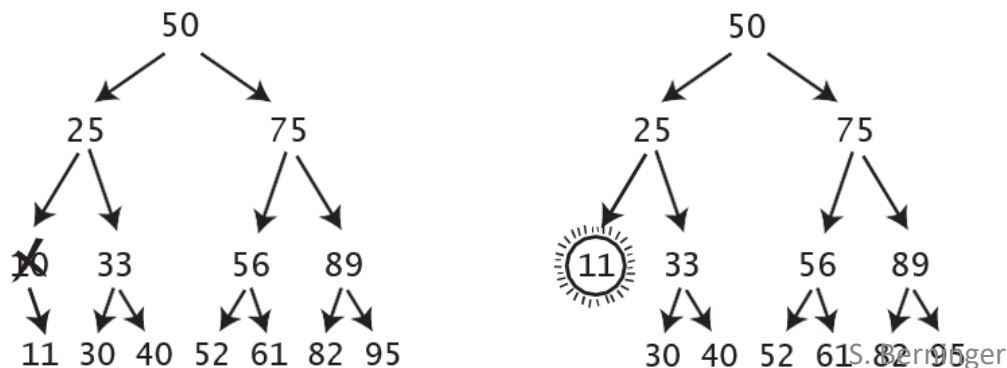
# Löschen

Die am wenigsten einfache Operation in einem binären Suchbaum!

Wir löschen die 4:



Wir löschen jetzt die 10: und setzen die 11 an die Stelle der 10:



Bis hierher folgt unser Löschalgorithmus folgenden Regeln:

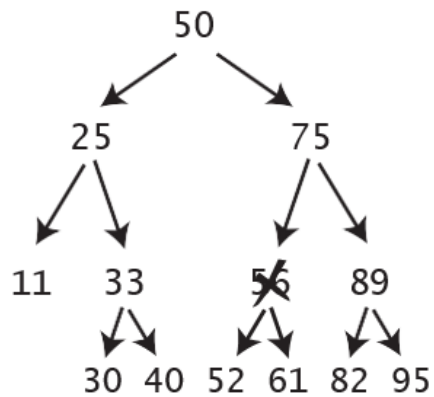
- hat der Knoten keine Kinder: einfach löschen
- hat der Knoten 1 Kind (Teilbaum), löschen und in die freie Stelle des gelöschten Knotens setzen



# Löschen

Komplexestes Szenario: Löschen eines Knotens mit 2 Kindern

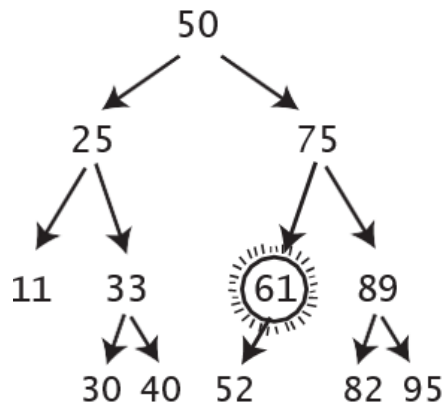
Löschen der 56:



Nächste Regel des Löschalgorithmus:

Hat der zu löschende Knoten 2 Kinder, ersetzt man den gelöschten Knoten durch den "Nachfolger"-Knoten. Dieser hat den *kleinsten aller Werte, die größer als der gelöschte Wert sind*.

Anders gesagt, ist der Nachfolgerknoten der nächstgrößere Knoten im Baum (der kleinste Knoten im rechten Teilbaum).



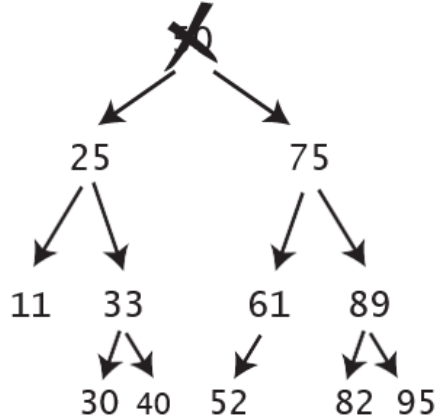
Wie findet der Algorithmus den Nachfolgerknoten?

Prüfe den rechten Kindknoten des gelöschten Knotens, und dann prüfe weiter den linken Kindknoten jeden folgenden Kindes, bis keine Kindknoten mehr vorhanden sind.

Der Wert auf der untersten Ebene ist der Nachfolgerknoten.

# Löschen

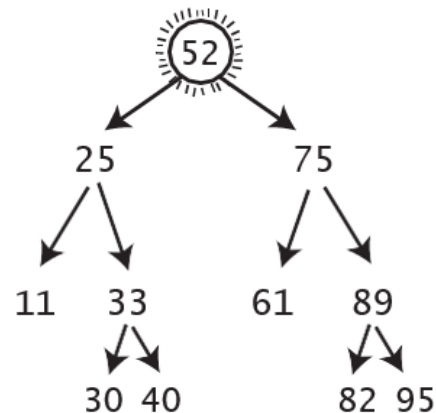
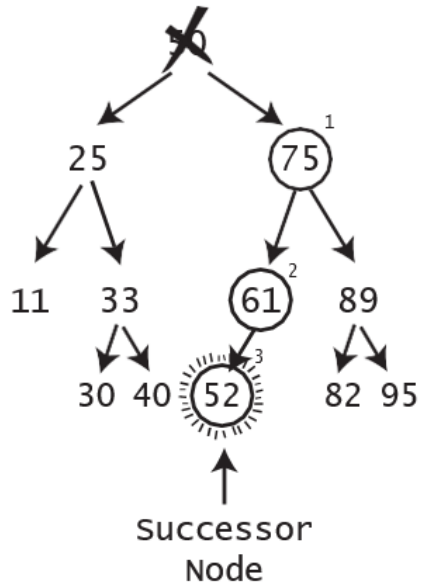
Wurzelknoten löschen:



Nachfolgerknoten finden:

Zunächst den rechten Kindknoten prüfen, Suche bei dem linken Kindknoten fortsetzen, bis zu einem Knoten, der keinen linken Kindknoten mehr hat.

Durch diesen Knoten ersetzen wir den gelöschten Knoten:



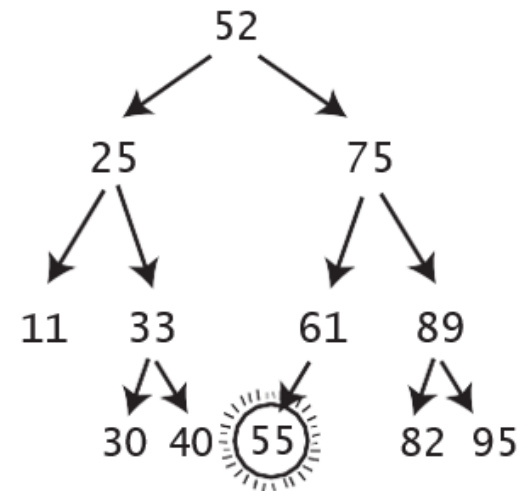
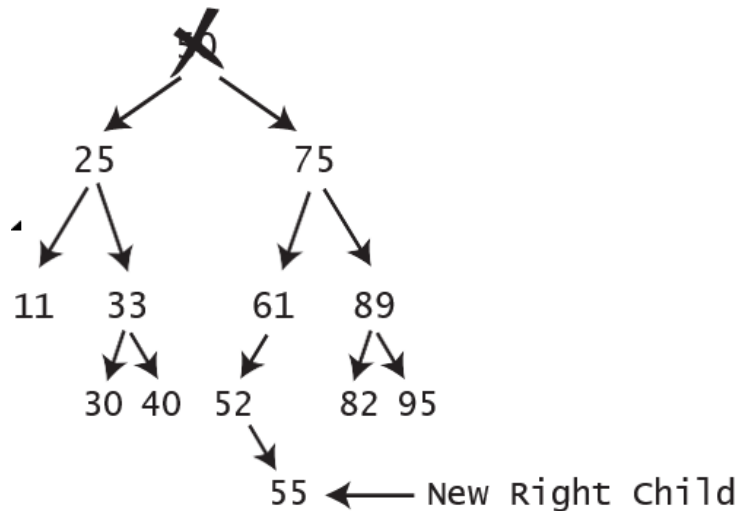
# Löschen

Bislang nicht betrachteter Fall: der Nachfolgerknoten hat nur einen rechten Kindknoten

Die 52 bekommt vor dem Löschen der Wurzel einen rechten Kindknoten:

Weitere Regel des Löschalgorithmus :

Hat der Nachfolgerknoten nur ein rechtes Kind, setzen wir (nachdem wir den gelöschten Knoten durch den Nachfolger ersetzt haben) das rechte Kind des Nachfolgerknotens an die frei gewordene Stelle des Nachfolgerknotens (als linken Kindknoten des ehemaligen Elternknotens des Nachfolgerknotens)



# Löschen

Alle Schritte des Löschalgorithmus im binären Suchbaum zusammen sind:

- Hat der zu löschende Knoten **keine Kinder**: einfach löschen.
- Hat der Knoten **ein Kind**: löschen, und das Kind an die Stelle des gelöschten Knotens setzen.
- Hat der Knoten **2 Kinder**: den zu löschenden Knoten durch den Nachfolgerknoten ersetzen.  
Der Nachfolgerknoten ist der Kindknoten, dessen Wert der kleinste aller größeren Werte des Baumes als der Wert des zu löschenden Knotens ist.  
\*Um den Nachfolgerknoten zu finden: das rechte Kind des zu löschenden Knotens prüfen, und dann die Suche fortsetzen bei dessen linken Kindknoten, bis wir einen Knoten erreichen, der keinen linken Kindknoten mehr hat.  
Durch diesen Knoten den gelöschten Knoten ersetzen.  
  
Hat der Nachfolgerknoten ein rechtes Kind, dann setzen wir das rechte Kind des Nachfolgerknotens an die freigewordene Stelle des Nachfolgerknotens.

# Löschen: Implementierung

```
// hebt Wert von minNode (Nachfolger) in zu loeschenden Knoten
struct treeNode *lift (struct treeNode *pNode, struct treeNode *pNodeToDelete)
{
    if (NULL != pNode->pLeft)
    { // linken Teilbaum rekursiv hinabsteigen
        pNode->pLeft = lift(pNode->pLeft, pNodeToDelete);
        return pNode;
    }
    else
    {
        // kein weiterer linker Knoten mehr
        pNodeToDelete->data = pNode->data; // Wert tauschen
        return pNode->pRight; // rechten Knoten oder Null zurückgeben
    }
}
```

```
struct treeNode * delete (int valueToDelete, struct treeNode * pNode)
{
    if (NULL == pNode) return NULL;
    else
    {
        // Element finden
        if (valueToDelete < pNode->data) // links weitersuchen
        {
            pNode->pLeft = delete(valueToDelete, pNode->pLeft);
            return pNode;
        }
        else
        {
            if (valueToDelete > pNode->data) // rechts weitersuchen
            {
                pNode->pRight = delete(valueToDelete, pNode->pRight);
                return pNode;
            }
            else
            {
                if (valueToDelete == pNode->data)
                {
                    // Schritt 2: Element gefunden: loeschen!
                    if (NULL == pNode->pLeft) return pNode->pRight;
                    // kein linker Teilbaum: rechten Teilbaum nach oben verketteten
                    else
                    {
                        if (NULL == pNode->pRight) return pNode->pLeft;
                        // kein rechter Teilbaum: linken TB nach oben verketteten
                        else
                        {
                            // Wert hat 2 Kinder: durch Nachfolger ersetzen
                            pNode->pRight = lift(pNode->pRight, pNode);
                            return pNode;
                        }
                    }
                }
            }
        }
    }
}
```

# Löschen

Löschen aus binären Suchbäumen: Komplexität von  **$O(\log N)$**  - wie Suche und Einfügen

- erfordert: Suche plus ein paar extra Schritte (um die nachfolgenden Kinder umzuhängen)

Das Löschen eines Wertes aus einem sortierten Array:  $O(N)$  wegen des Verschiebens der Elemente nach links.

# Binäre Suchbäume in Aktion

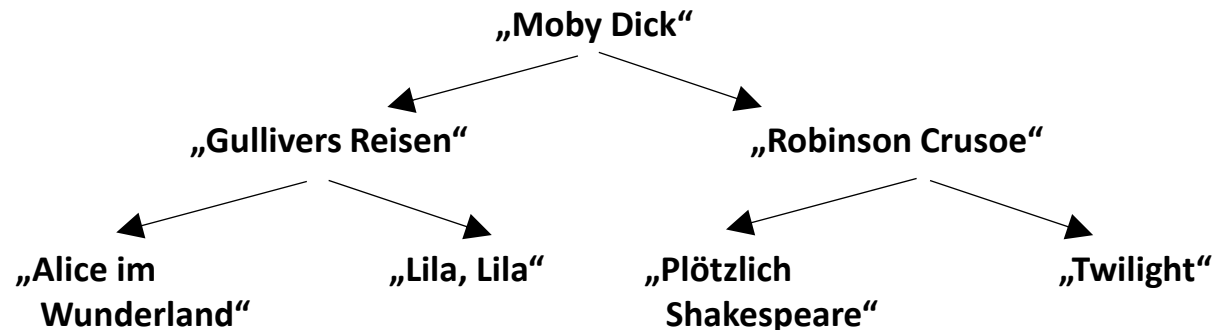
Applikation, die eine Liste von Buchtiteln verwaltet, Funktionalität:

- Buchtitel in alphabetischer Reihenfolge drucken können
- Änderungen an der Liste erlauben
- dem Benutzer die Suche nach einem Titel in der Liste erlauben

Würde sich unsere Liste nicht hochfrequent ändern: sortiertes Array wäre die passende Datenstruktur

Aber: App soll viele Änderungen in Echtzeit verarbeiten können.

Da unsere Liste Millionen von Titeln haben wird: besser einen binären Suchbaum verwenden!



# Binäre Suchbäume in Aktion

Buchtitel in alphabetischer Reihenfolge ausdrucken können:

1. Möglichkeit nötig, jeden Knoten des Baumes zu besuchen (*“Traversieren”* der Datenstruktur)
2. Sicherstellen, den Baum in alphabetisch aufsteigender Ordnung zu traversieren, um die Liste in dieser Reihenfolge ausdrucken zu können: **“Inorder** Traversierung”

Rekursion eignet sich grossartig für Traversierung.

Schritte der Funktion *“traverse()”*:

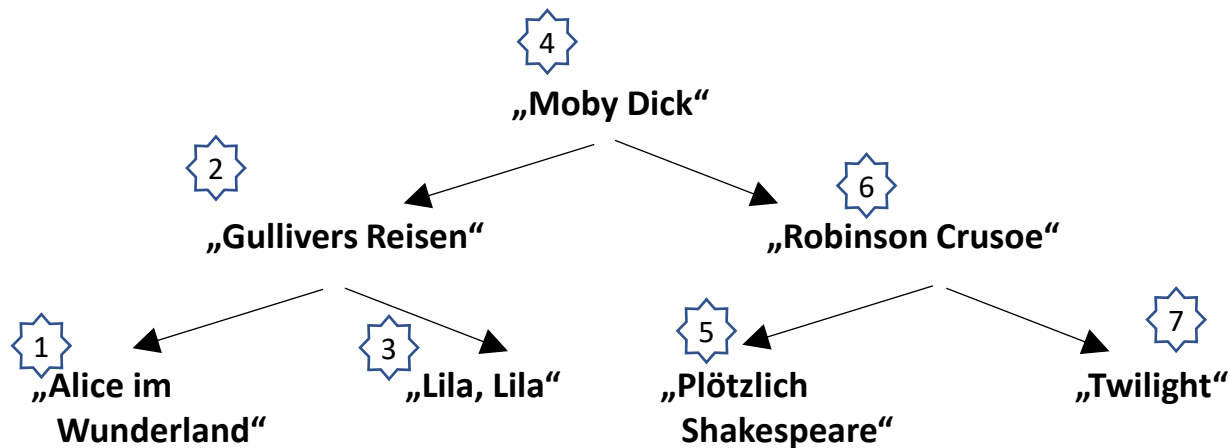
1. Ruft sich selbst rekursiv für das linke Kind des Knotens auf. Wird dadurch so lange aufgerufen, bis sie auf einen Knoten trifft, der kein linkes Kind hat (der Weg dorthin geht auf den Stack).
2. *“Besucht”* den Knoten. (...in unserer Buchtitel-App drucken wir den Wert des Knotens hier aus...)
3. Ruft sich selbst rekursiv für das rechte Kind des Knotens auf. Wird dadurch so lange aufgerufen, bis sie auf einen Knoten trifft, der kein rechtes Kind hat.



# Binäre Suchbäume in Aktion

Abbruchkriterium: wird *traverse()* für einen Teilbaum aufgerufen, der nicht existiert:  
Rückkehr, ohne etwas Weiteres zu tun.

Kehrt *traverse()* im Knoten „Moby Dick“ aus dem Aufruf des rechten Teilbaums zurück, haben wir alle Knoten des Baums besucht.



```
void traverse_and_print(struct treeNode *pNode)
{
    if (NULL == pNode)    return;
    traverse_and_print (pNode->pLeft);
    printf ("%d\n",pNode->data);
    traverse_and_print (pNode->pRight);
}
```

# Zusammenfassung

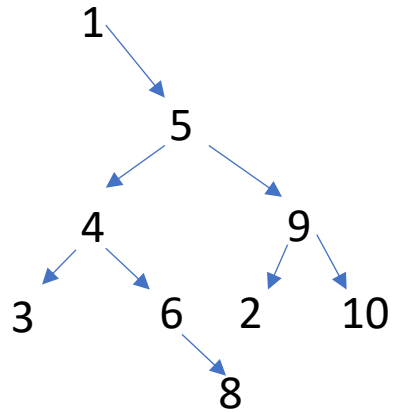
Der binäre Suchbaum ist eine mächtige knotenbasierte Datenstruktur, die die Reihenfolge bewahrt, und gleichzeitig schnelles Suchen, Einfügen und Löschen erlaubt.

Der binäre Suchbaum ist nur eine von vielen Baumarten.

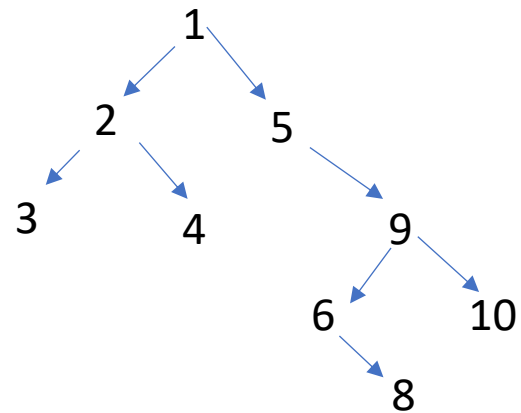
# Übung:

1. Fügen Sie die folgende Zahlenfolge (in der gegebenen Reihenfolge) einem leeren binären Suchbaum hinzu: [1, 5, 9, 2, 4, 10, 6, 3, 8]. Welche Lösung ist korrekt?

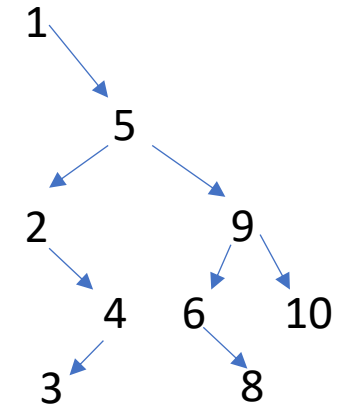
A



B



C



2. Wenn ein ausgeglichener binärer Suchbaum 1.000 Werte enthält, wieviele Schritte sind dann maximal nötig, um einen Wert zu finden?

A: 1

B: 10

C: 100

D: 500

E: 1000

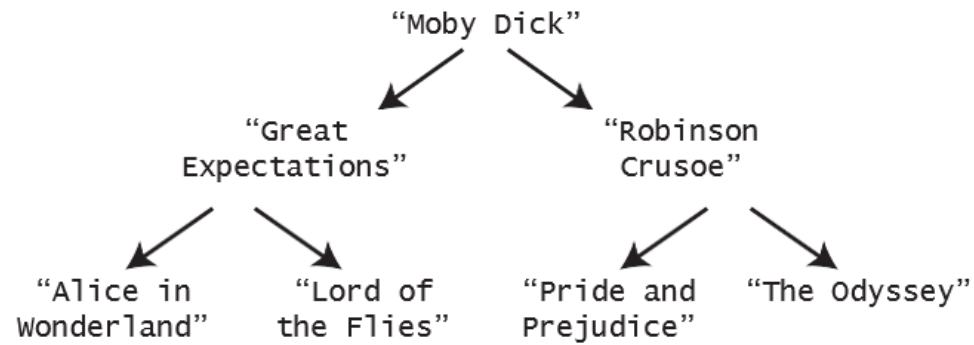
3. Wo steht der größte Wert in einem binären Suchbaum?

A: ganz oben

B: ganz rechts außen

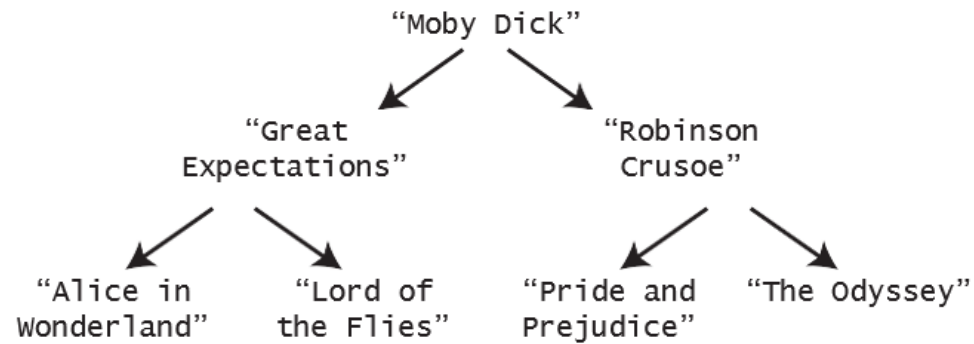
C: auf der untersten Ebene rechts

4. In welcher Reihenfolge werden die Titel mit der Preorder Traversierung erreicht?



- A: Moby - Great - Alice - Lord - Robinson - Pride - Odyssey  
B: Alice - Great - Lord - Moby - Pride - Robinson - Odyssey  
C: Alice- Lord - Great - Pride - Odyssey - Robinson - Moby

5. Geben Sie für den Beispielbaum die Reihenfolge an, in der die Titel mit Postorder Traversierung erreicht werden.



A: Moby - Great - Alice - Lord - Robinson - Pride - Odyssey

B: Alice - Great - Lord - Moby - Pride - Robinson - Odyssey

C: Alice- Lord - Great - Pride - Odyssey - Robinson - Moby

# Prioritäten setzen mit Heaps

Es gibt viele weitere Arten von Bäumen mit eigenen Vor- und Nachteilen:  
-> welchen für welche Situation?

Der Heap ist ein spezieller Baum –  
besonders geeignet, wenn wir das größte oder kleinste Element in einem Datensatz im Auge behalten wollen.



# Anwendungsbeispiel eines Heaps: Prioritäts-Queue

- Die Queue ist eine Datenstruktur, deren Elemente nach dem FIFO-Prinzip bearbeitet werden.



- Prioritäts-Queue: Entnehmen wie in klassischer Queue (an der Spitze der Queue), aber das Einfügen erfolgt wie in ein sortiertes Array (Daten bleiben stets sortiert).

Klassisches Beispiel: Triage in einer Notaufnahme. Die Patienten werden nicht in der Reihenfolge ihrer Aufnahme behandelt, sondern beim Eintreffen einsortiert nach Dringlichkeit.

- Annahme: Der Schweregrad der Erkrankung der Patienten wird auf einer Skala von 1 bis 10 eingeteilt, mit 10 als dem schwersten Grad.

Prioritäts-Queue:


Patient C - Severity: 10
Patient A - Severity: 6
Patient B - Severity: 4
Patient D - Severity: 2

↑ front of priority queue

# Anwendungsbeispiel: Prioritäts-Queue

- „Der Nächste bitte“ ruft immer den Patienten an der Spitze der Queue auf, weil er der mit der höchsten Dringlichkeit ist (im Beispiel Patient C)
- Ein eintreffender Patient E mit dem Schweregrad 3 wird hier einsortiert:

Patient C - Severity: 10
Patient A - Severity: 6
Patient B - Severity: 4
Patient E - Severity: 3
Patient D - Severity: 2



- Prioritäts-Queue ist ein abstrakter Datentyp – wird implementiert auf Basis anderer, fundamentaler Datentypen, z.B. einem einfachen sortierten Array, unter folgenden Randbedingungen:
  - Beim/ nach dem Einsortieren stellen wir die Sortierreihenfolge immer wieder her
  - Daten können nur am Ende des Arrays (Spitze der Queue) entnommen werden

# Anwendungsbeispiel: Prioritäts-Queue

Effizienz-Analyse:

- 2 primäre Operationen: Entnehmen und Einfügen

Entnehmen:

- Entnehmen am Anfang eines Arrays:  $O(N)$ , weil alle Elemente für das Schließen der Lücke verschoben werden müssen
- Entnehmen am Ende eines Arrays:  $O(1)$  – deshalb Entnahme (max. Prio) am Ende

Einsortieren:

- Einfügen in ein sortiertes Array ist  $O(N)$   
(mit allen Elementen muss verglichen werden – und bei früher Einfügung alle anderen verschoben)

In Summe:  $O(N)$  – bei vielen Elementen eine unwillkommene Verzögerung

Bessere Datenstruktur für Prioritäts-Queues als Sortiertes Array: **HEAP!**

# Binärer Heap

Es gibt 7 verschiedene Arten von Heaps – Fokus ist hier auf ***Binären Heaps***

= spezieller binärer Baum, aber kein binärer Suchbaum!

2 Typen binärer Heaps: Max-Heap und Min-Heap

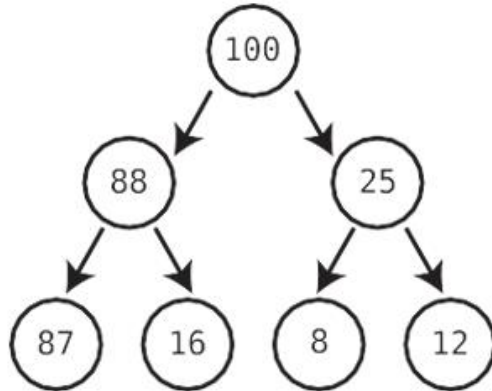
Binärer Heap: folgt folgenden Bedingungen (*heap condition*) :

- a) Der Wert jeden Knotens muss gleich oder größer sein als die Werte seiner Kinder (Max-Heap)  
oder jeder Wert gleich oder kleiner (Min-Heap) als die Werte seiner Kinder
- b) Der Baum muss vollständig aufgefüllt sein

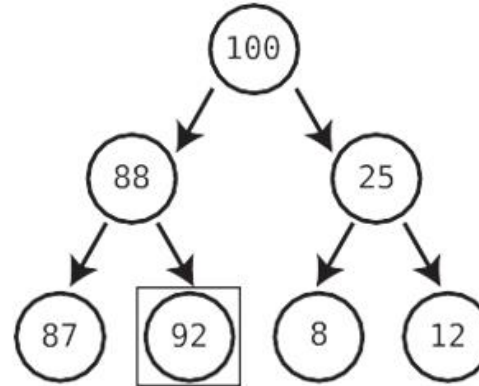
# Binärer Heap: Heap condition

- a) Der Wert jeden Knotens muss größer sein als die Werte seiner Kinder (Max-Heap) oder jeder Wert kleiner (Min-Heap)
- b) Der Baum muss heap-vollständig sein

zu a) Heap condition erfüllt:



Heap condition nicht erfüllt:



Heap ist anders strukturiert als Binärer Suchbaum: dort ist jedes rechte Kind größer als sein Vorgänger.  
Im binären Heap ist kein Kind größer als sein Vorgänger!

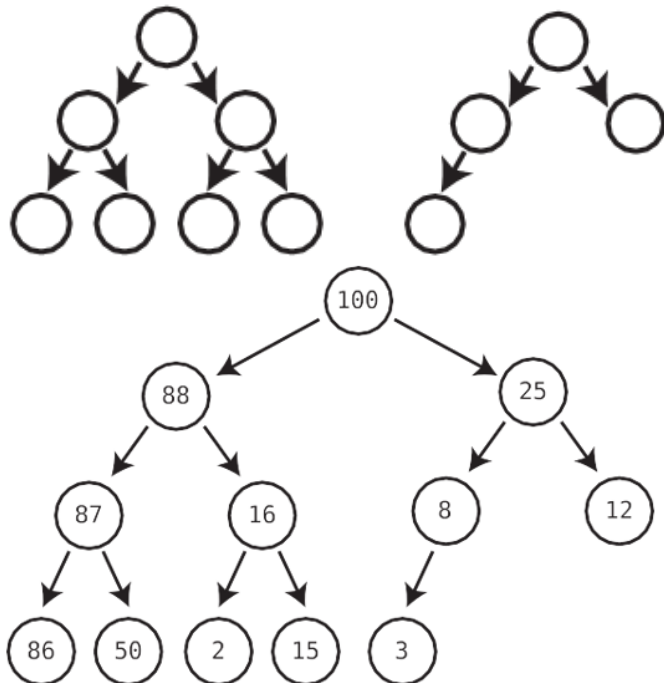
Ein binärer Suchbaum ist kein Heap!

# Binärer Heap: Heap-Vollständigkeit

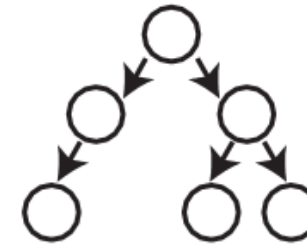
zu b) Heap-Vollständigkeit:

Ein Baum ist heap-vollständig, wenn beim Lesen der Knoten einer Ebene von links nach rechts alle Knoten vorhanden sind (Ausnahmen sind nur auf der untersten Ebene bei den letzten Elementen erlaubt).

Heap-vollständige Bäume:



Heap-unvollständiger Baum:



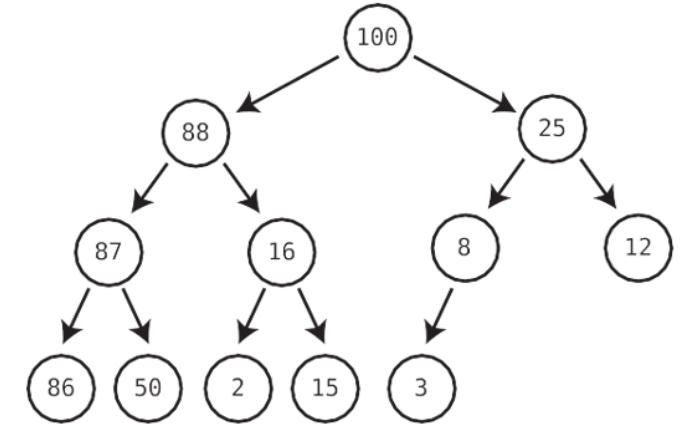
# Binärer Heap: Eigenschaften

Die Sortierung eines Heap ist nutzlos für die Suche im Heap!

Beispiel: Suche nach „3“ und Start bei 100: rechts oder links weitersuchen?

Binärer Suchbaum: „3“ müßte unter den linken Knoten sein

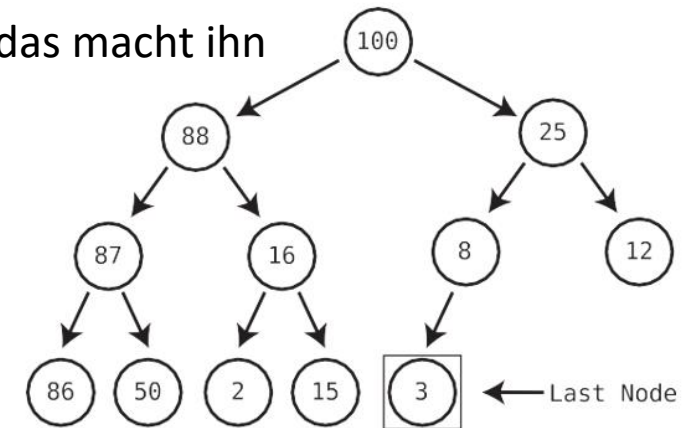
Binärer Heap: „3“ ist irgendein Kind unterhalb von „100“



Heaps sind „**Schwach geordnet**“ gegenüber „Stark geordnetem“ Binären Suchbaum

Der **Wurzelknoten** ist immer der **größte (oder kleinste) Knoten** im gesamten Baum (das macht ihn besonders geeignet für Priority queues).

Der Heap hat einen „**letzten Knoten**“ (rechtster Knoten in der untersten Ebene).

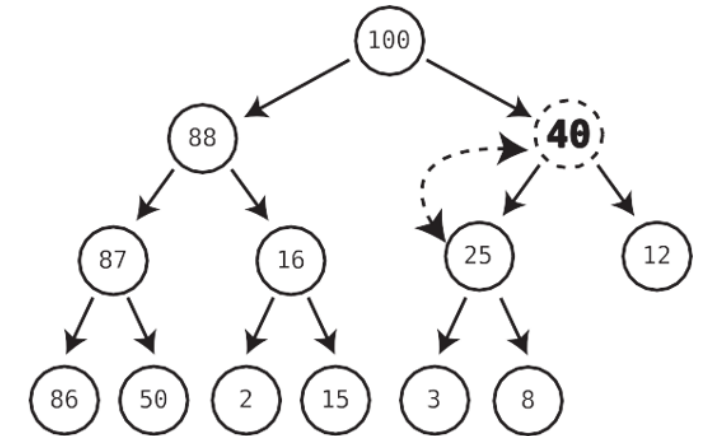
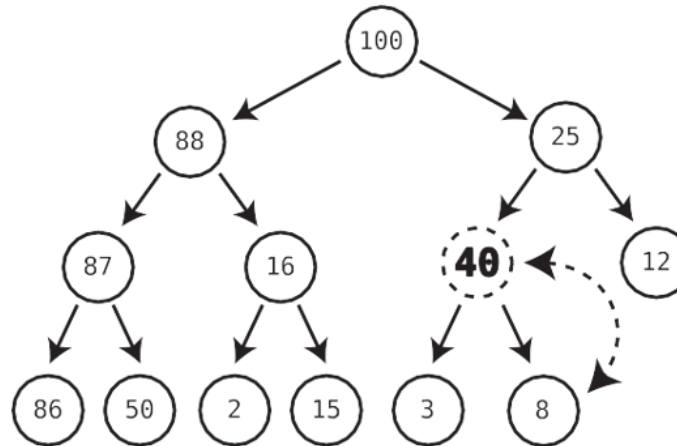
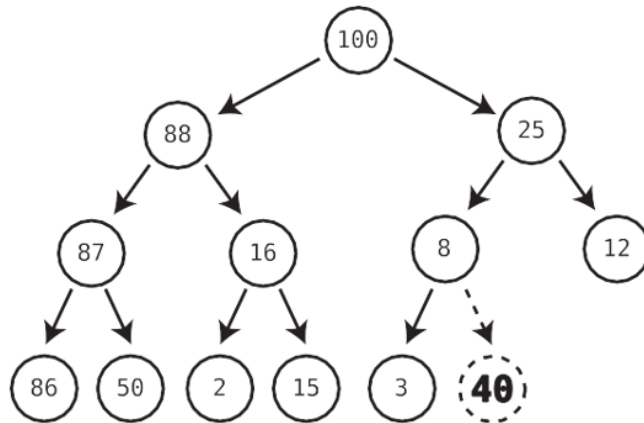


# Binärer Heap: Einfügen

Algorithmus:

1. Wir erzeugen zunächst einen neuen letzten Knoten mit dem einzufügenden Wert. Das ist der nächste freie rechte Platz auf der untersten Ebene (oder auf einer neuen Ebene).
2. Danach vergleichen wir diesen neuen Knoten mit seinem Vater-Knoten.
3. Ist der neue Knoten größer als sein Vaterknoten, vertauschen wir die Beiden, und setzen fort bei Schritt 2. Ansonsten sind wir fertig.

Einfügen von #40:



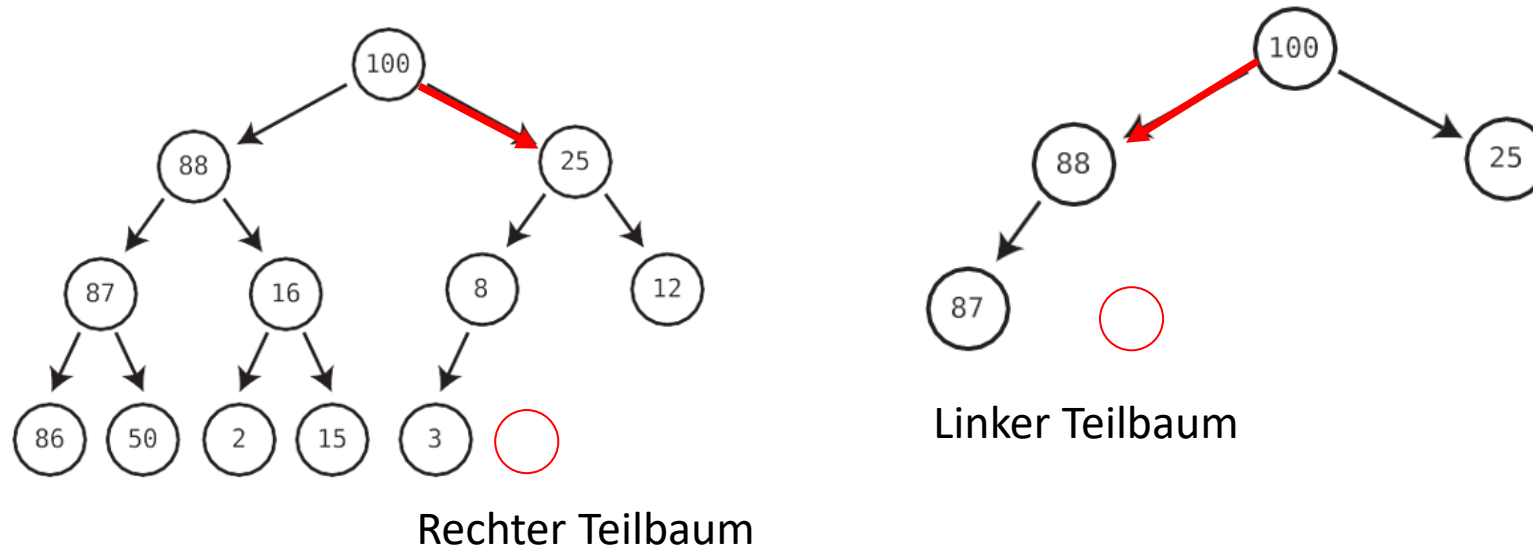


# Binärer Heap: Einfügen

Prozess des Verschieben des Knotens im Baum nach oben nennt sich „Versickern“.

Effizienz des binären Heap:  $O(\log N)$  (Binärer Baum mit  $N$  Knoten hat  $\log(N)$  Ebenen)

Notwendig: Zuerst Suche nach dem letzten Knoten – algorithmisch!



-> jeder Knoten muss besucht werden (wie auch beim Entnehmen – s. folgende Folien)!

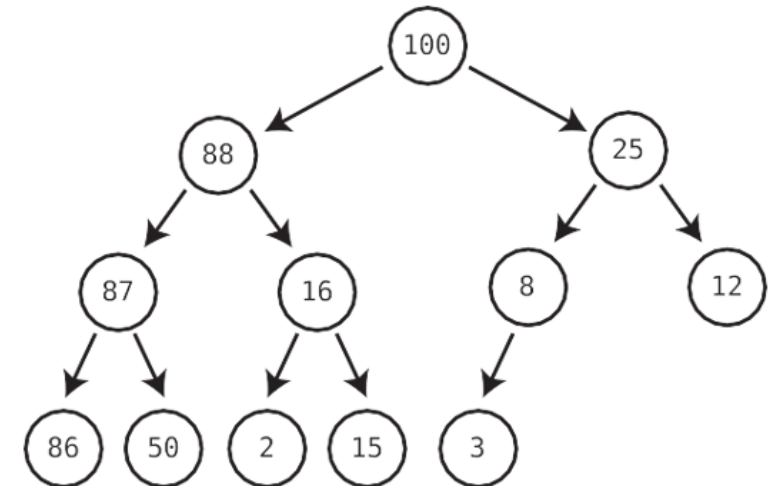
# Binärer Heap: Entnehmen

Wir entnehmen immer nur den Wurzelknoten (mit höchster Priorität)!

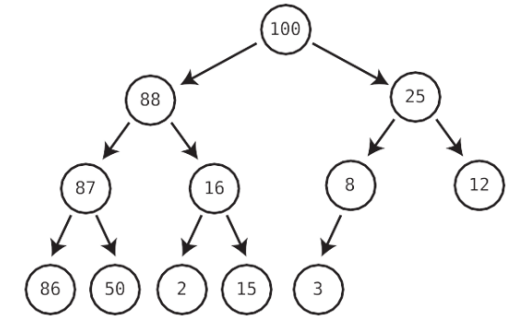
Algorithmus:

1. Entnimm den Wurzelknoten und ersetze ihn durch den letzten Knoten.
2. Versickere den neuen Wurzelknoten an seinen korrekten Platz ("Versickern nach unten" wird später erklärt).

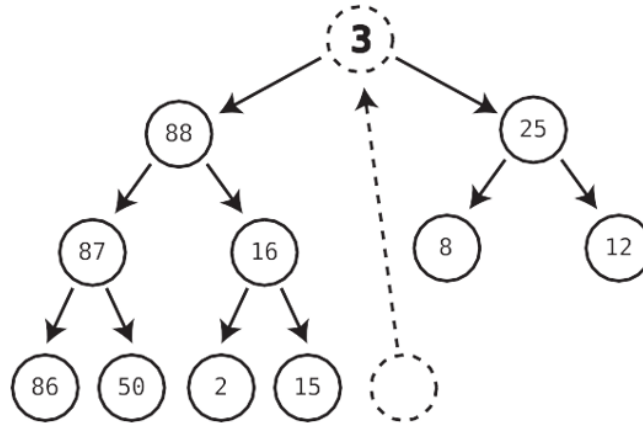
Wir entnehmen im Folgenden den Wurzelknoten des folgenden Heaps:



# Binärer Heap: Entnehmen



zu 1.: Wir ersetzen den Wurzelknoten durch den letzten Knoten:



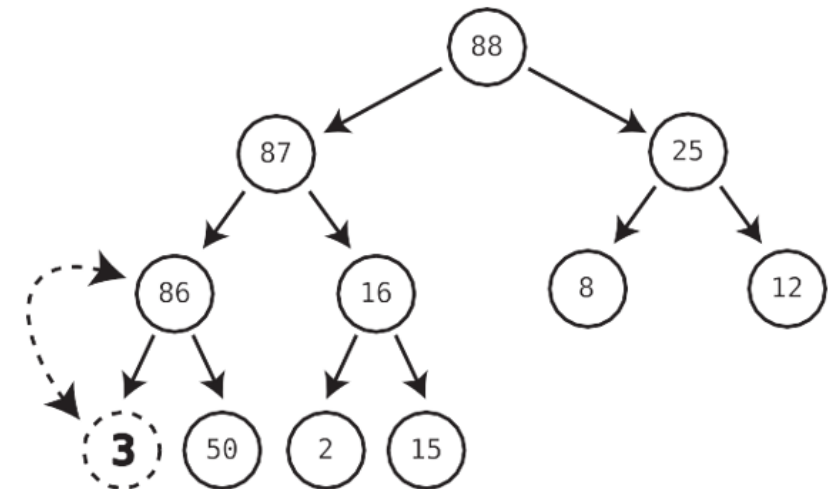
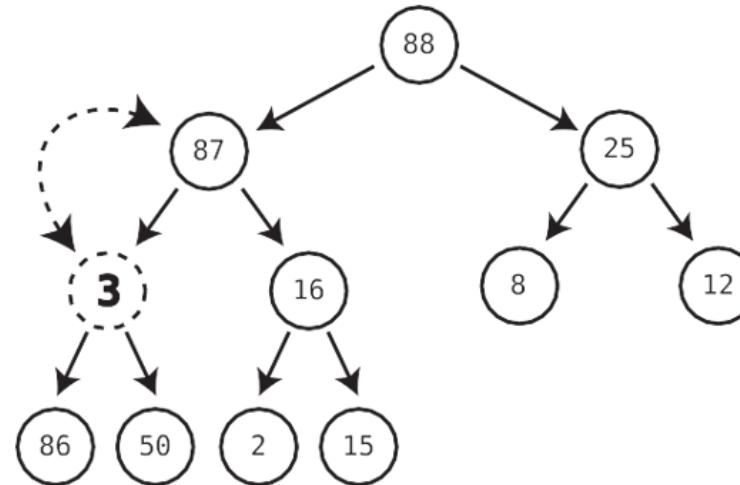
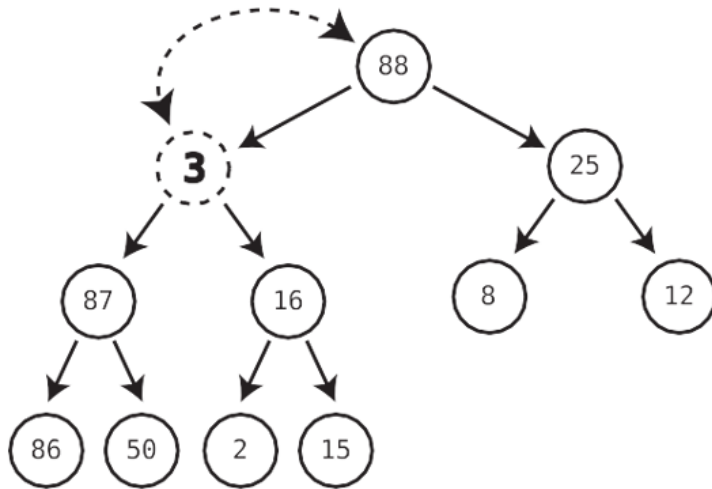
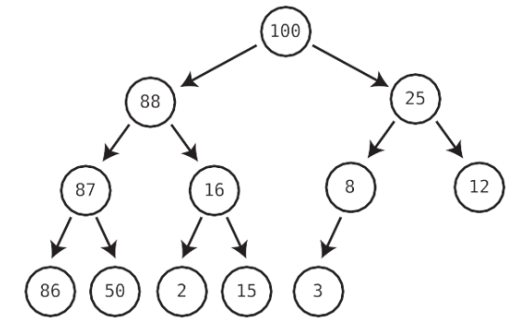
zu 2.: Wir müssen nun die 3 (den “Versickerungsknoten”) nach unten versickern:

- wir haben auf jeder Ebene 2 mögliche Richtungen: vertauschen mit dem linken oder rechten Kind
  - a) Wir wählen das größere der beiden Kinder aus
  - b) Wir prüfen, ob dieses Kind größer ist als der Versickerungsknoten.  
Wenn ja, tauschen wir die Knoten, und gehen wieder zu a)

Wenn nicht, sind wir fertig.

Dann hat der Versickerungsknoten keine Kinder mehr, die größer sind als er selbst.

# Binärer Heap: Entnehmen



Wie die Zeitkomplexität des Einfügens ist auch die des Löschens aus einem Heap  $O(\log N)$ .

# Heap vs. Sortiertes Array

Welche Datenstruktur benutzen wir nun für Priority queues?

	Sortiertes Array	Heap
Einfügen	$O(N)$	$O(\log N)$
Entnehmen	$O(1)$	$O(\log N)$

Warum sind Heaps die bessere Wahl? Weil  $O(1)$  extrem schnell ist,  $O(\log N)$  immer noch ziemlich schnell. Und  $O(N)$  eher langsam.

	Sortiertes Array	Heap	
Einfügen	Langsam	Ziemlich schnell	Priority queues benötigen gleichviel Einfügungen wie Entnahmen -> beide Operationen müssen schnell sein.
Entnehmen	Extrem schnell	Ziemlich schnell	Ist eine der beiden Operationen langsam, verzögert sie die gesamte Queue.
			<b>Der Heap garantiert einen „ziemlich schnellen“ Betrieb.</b>

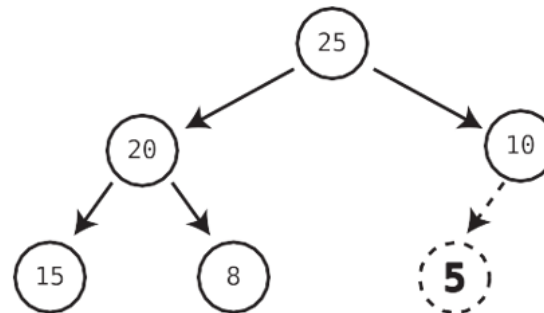
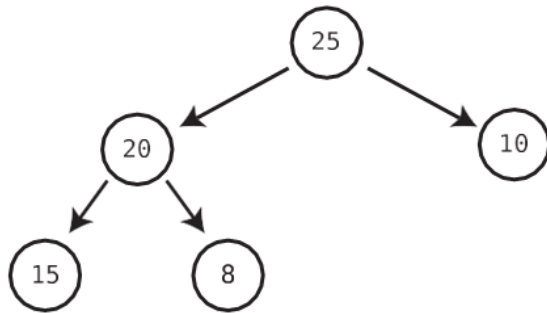
# Finden des letzten Knotens

Wie kann der letzte Knoten gefunden werden, um:

- beim Entnehmen den Wurzelknoten zu ersetzen
- einen neuen Knoten als letzten Knoten einzusetzen?

Heap muss vollständig bleiben!

Erneuter Blick auf's Einfügen (Knoten mit Wert 5):



Jede Alternative wäre unausgeglichen.  
Nur die Vollständigkeit erlaubt uns,  $O(\log N)$  zu erreichen!

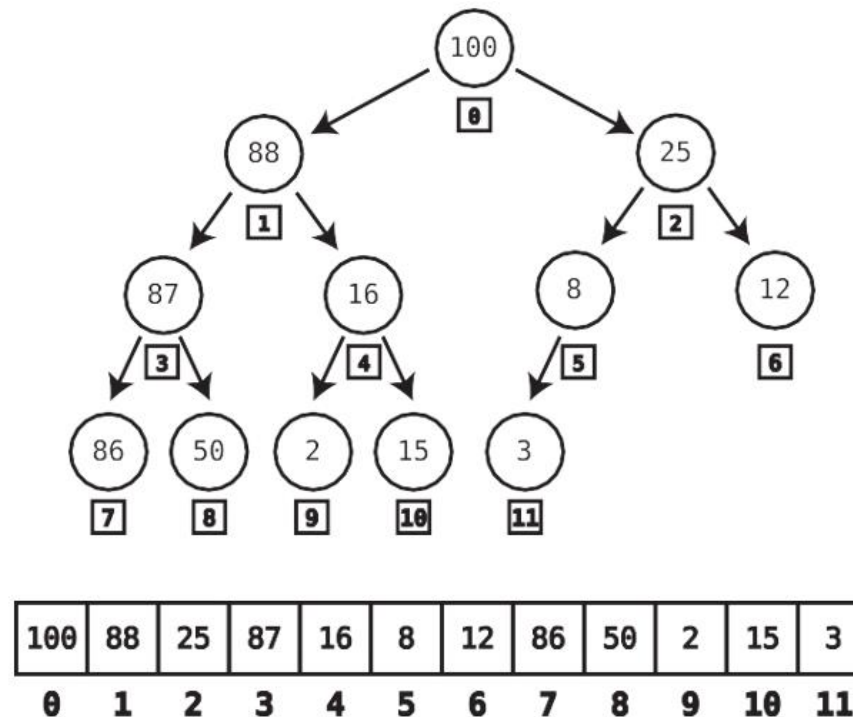
# Finden des letzten Knotens: Arrays als Heaps

Wir brauchen einen effizienten Weg, den letzten Knoten zu finden!

Lösung: Wir implementieren Heaps als (unsortierte) Arrays!

Bisher nutzen wir für den Heap Bäume auf Basis verkettete Listen. Wir können aber auch Arrays verwenden (als Basis für den Heap als abstrakten Datentyp):

Nr. des Knotens im Baum ist Index des Arrays!



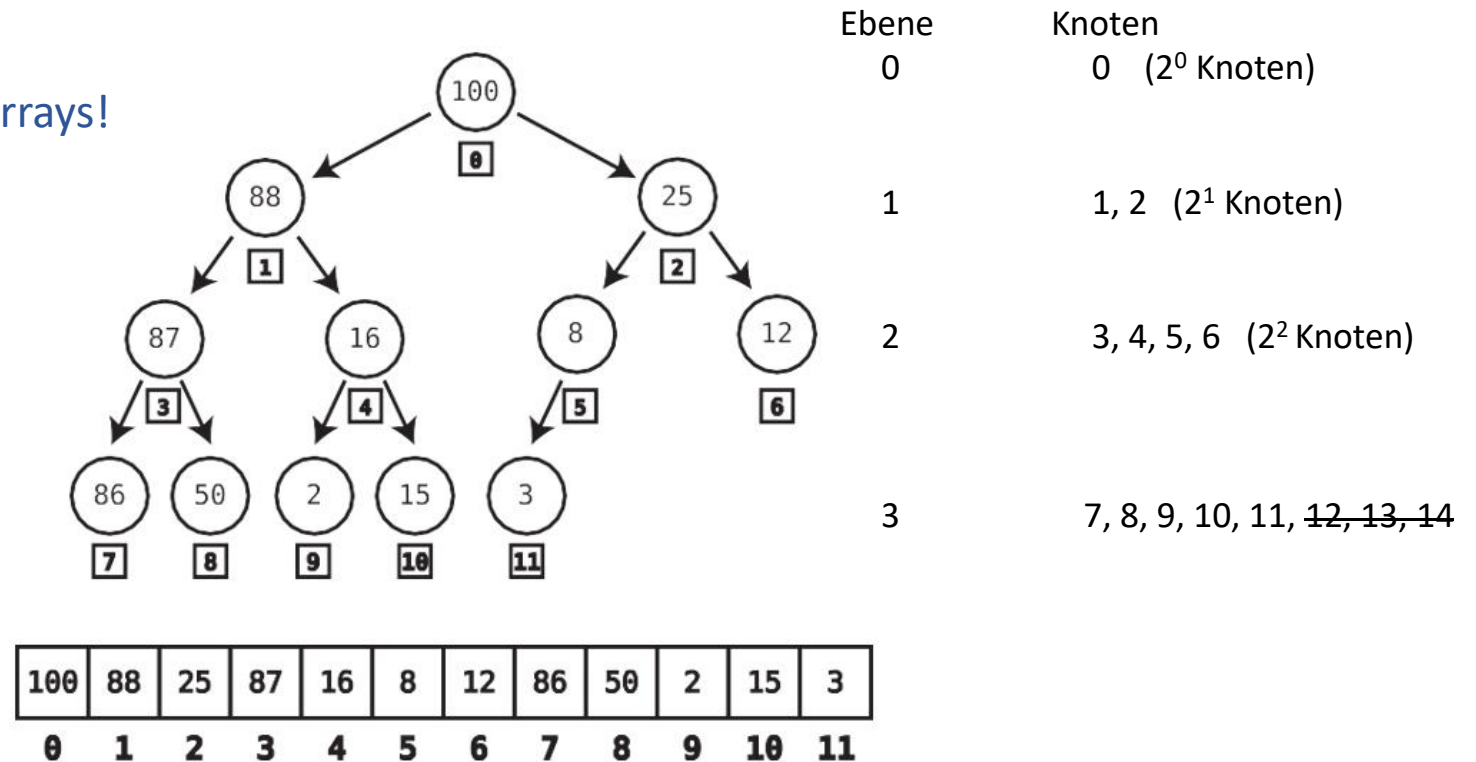
# Finden des letzten Knotens: Arrays als Heaps

Ein binärer vollständiger Heap hat auf jeder Ebene  $n$  genau  $2^n$  Knoten – bis auf die letzte Ebene, die mit dem Ende des Arrays endet.

Der Wurzelknoten hat immer Index 0.

Der letzte Knoten ist das letzte Element des Arrays!

Auffinden und anfügen ist dadurch einfach...





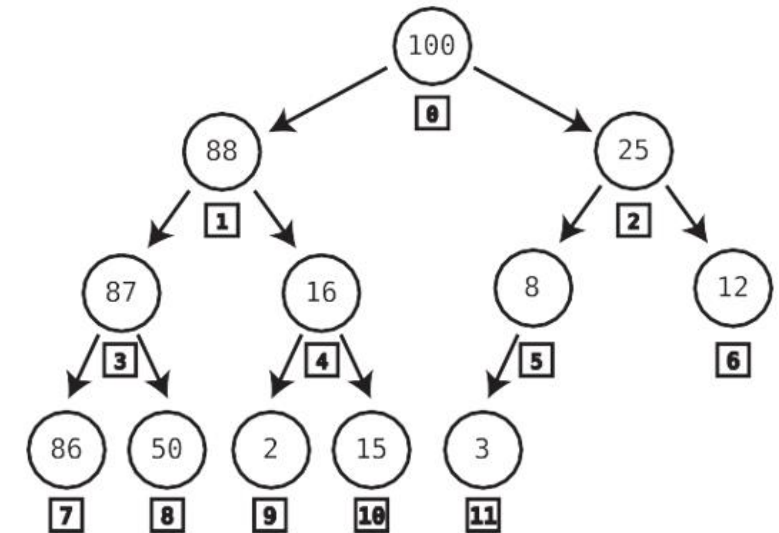
```
// Deklaration des Typs Heap
typedef struct
{
    bool empty;
    int last; // Index des letzten Elements
    int * values; // Wertearray
} heap;

void heap_init (heap* h, int* storage)
{
    // leeres Array
    h->empty=true;
    h->last = 0; // letzter Knoten hat Index 0
    h->values = storage;
    // Speicheradresse des Arrays
}
```

# Heap-Implementierung

Die Knoten jeder Ebene sind:

	Indices
Ebene $n$	$2 \cdot (2^{n-1} - 1) + 1 \dots 2 \cdot (2^n - 1)$
0	0 ... 0
1	1 ... 2
2	3 ... 6
3	7 ... 14
4	15 ... 30
5	31 ... 62
...	



100	88	25	87	16	8	12	86	50	2	15	3
0	1	2	3	4	5	6	7	8	9	10	11

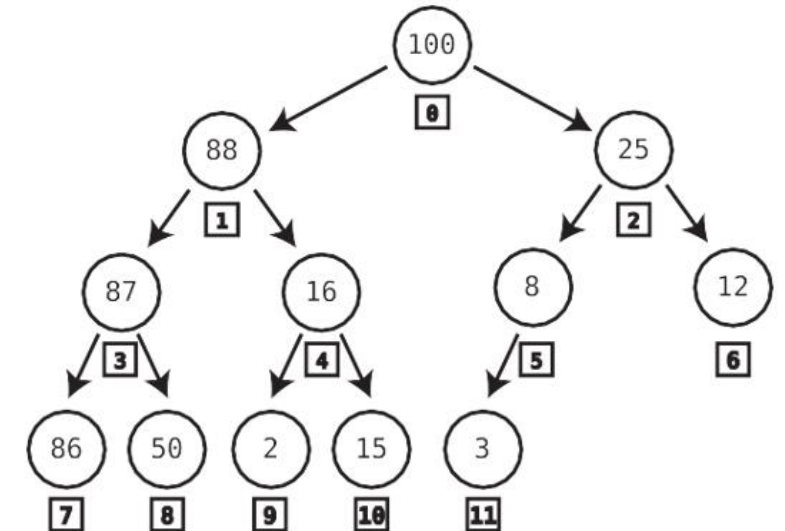
# Heap-Implementierung

Wie traversieren?

Index des linken Kindes eines Knotens:  $(\text{elternIndex} * 2) + 1$

Index des rechten Kindes eines Knotens:  $(\text{elternIndex} * 2) + 2$

Index des Vorfahrens eines Knotens :  $(\text{kindIndex} - 1) / 2$   
(2 Kinder haben gleichen Elternindex)



100	88	25	87	16	8	12	86	50	2	15	3
0	1	2	3	4	5	6	7	8	9	10	11



# Heap-Implementierung: Hilfsfunktionen

```
int LeftChildIndex (heap* h, int n)
{
    return (n*2 + 1);
}
```

```
int RightChildIndex (heap* h, int n)
{
    return (n*2 + 1 + 1);
}
```

```
int ParentIndex (heap* h, int n)
{
    return ((n-1) / 2); // cutting the remainder
}
```

# Heap-Implementierung: Insert

```
int Insert (heap* h, int value)
{
    if (true == h->empty)
    {
        h->empty=false;
    }
    else
    {
        // Verlaengern um 1 Element
        if ( NULL == realloc (h->values, sizeof (int)))
            return EXIT_FAILURE;
        h->last++;
    }
    h->values[h->last]=value; // neues letztes Element belegen
    Max_heap_bubble_up (h, h->last);
    return EXIT_SUCCESS;
}
```

```
void SwapIndices (heap* h, int n, int m)
{
    // swaps index n with m
    int tmp = h->values[n];
    h->values[n] = h->values[m];
    h->values[m] = tmp;
}
```

```
void Max_heap_bubble_up (heap* h, int n)
{
    int parent;
    while (n > 0) // 0: Wurzelindex
    {
        parent = ParentIndex (h, n);
        if (h->values[parent] > h->values[n]) break;
        // Parent > Child
        SwapIndices (h, parent, n);
        n = parent;
    }
}
```



# Heap-Implementierung: Extract

```
// not called if h->empty == true!  
int Max_heap_extract (heap* h)  
{  
    // Wurzel entnehmen: Index 0  
    return Max_heap_delete(h, 0);  
}
```

# Heap-Implementierung: Delete

```
int Max_heap_delete (heap* h, int n)
{
    // n: Index, meist 0 (Wurzel)
    int value = h->values[n];

    h->values[n] = h->values[h->last]; // ersetzen durch letzten Knoten
    if (0 == h->last) h->empty=true;
    else h->last -= 1; // letzter Index
    if (n >= h->last) return value; // letzten oder vorletzten Wert gelöscht

    if ((n > 0) && (h->values[n] > h->values [(n-1)/2]))
        Max_heap_bubble_up (h, n); // nicht Wurzel, Vater ist kleiner
    else Max_heap_sift_down (h, n); // Wurzel oder Vater ist grösser/ gleich
    return value;
}
```

s. Insert():

```
void Max_heap_bubble_up (heap* h, int n)

void Max_heap_sift_down (heap* h, int n)
{
    int last = h->last;
    int max = n;
    while (1)
    {
        if (HasGreaterChild(h,n))
        {
            max=GreaterChildIndex(h,n);
            if (n == max) break;
            SwapIndices (h, max, n);
            n = max;
        }
        else break;
    }
}
```

# Heap-Implementierung: Delete

```
bool HasGreaterChild (heap* h, int n)
```

```
{
    bool state=false;
    if (((LeftChildIndex(h, n) <= h->last)
        && (h->values[LeftChildIndex(h,n)] > h->values[n]))
        || ((RightChildIndex(h, n) <= h->last)
            && (h->values[RightChildIndex(h, n)] > h->values[n])))
        state=true;
    return state;
}
```

```
int GreaterChildIndex (heap* h, int n)
```

```
{
    int right= RightChildIndex(h, n);
    int left= LeftChildIndex(h, n) ;
    if (right <= h->last)
    {
        if (h->values[right] > h->values[left])
            return right;
        else return left;
    }
    else return left;
}
```



# Heaps für Priority queues

Alle Operationen mit  $O(\log N)$  –  
die schwache Sortierung der Heaps kommt den Priority queues ideal entgegen!

Verschiedene Baumarten eignen sich besser oder schlechter für verschiedene Probleme!

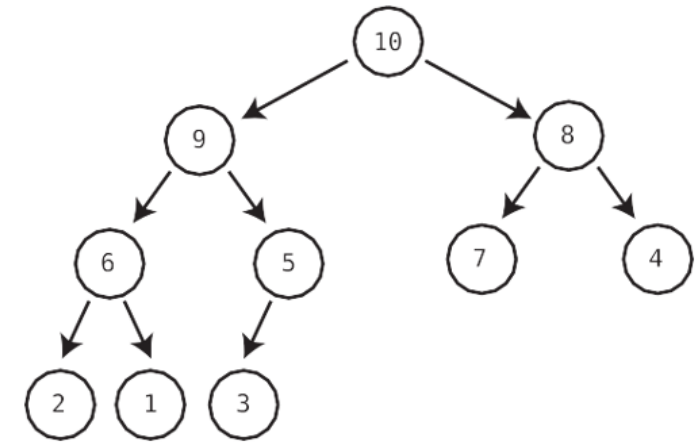
Binäre Suchbäume unterstützen eine schnelle Suche und minimieren die Kosten der Einfügung, während Heaps die perfekte Datenstruktur für Priority Queues sind.

-> Algorithmus und Datenstruktur müssen zusammenpassen, gut aufeinander abgestimmt sein!

In der nächsten Lektion sehen wir uns Bäume für die lexikale Analyse (Präfixbäume) näher an...

# Übung:

1. Wie sieht der folgende Heap aus, nachdem wir eine 11 eingefügt haben?



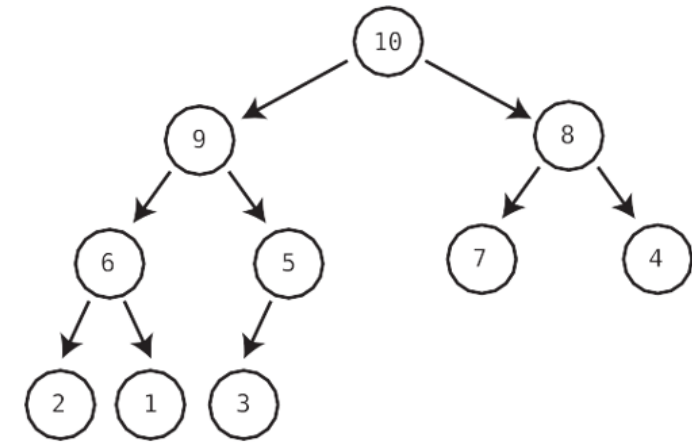
A: 11-10-9-8-6-5-7-4-2-1-3

B: 11-10-8-6-9-7-4-2-1-3-5

C: 10-9-8-6-5-7-4-2-1-3-11

# Übung:

2. Wie sieht der Heap aus, nachdem wir den Wurzelknoten gelöscht haben?



A: 9-8-6-5-7-4-2-1-3

B: 9-6-5-2-1-3 und 8-7-4

C: 9-6-8-3-5-7-4-2-1

3. Sie haben einen neuen Max-Heap erzeugt durch Einfügen von Zahlen in der folgenden Reihenfolge:  
55, 22, 34, 10, 2, 99, 68.

Wenn Sie diese jetzt aus dem Heap holen, einen nach dem anderen, und diese Zahlen in ein neues Array einfügen:  
in welcher Reihenfolge würden sie dann dort erscheinen?

A: In der Reihenfolge der Eingabe.

B: In perfekter absteigender Sortierung.

C: 99-68-55-34-22-10-2