# Datenstrukturen und Algorithmen

Vorlesung 7

# Überblick

- Vorige Woche:
  - Repräsentierungen für schwachbesetzte Matrix, Stack, Queue, Prioritätsschlange, Deque
  - Einfach verkettete Listen auf Arrays
- Heute betrachten wir:
  - Doppelt verkettete Listen auf Arrays
  - Heap

# Verkettete Listen auf Arrays - Beispiel

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| elems | | 78 | 11 | 6 | 59 | 19 | | 44 | | |
| next | 7 | 6 | 5 | -1 | 8 | 4 | 9 | 2 | 10 | -1 |

Head = 3

firstEmpty = 1

# SLLA – Repräsentierung

- Die Repräsentierung einer einfach verketteten Liste auf einem Array (SLLA) ist:

SLLA:
elems: TElem[]
next: Integer[]
cap: Integer
head: Integer
firstEmpty: Integer

# DLLA

- Ähnlich kann man auch eine doppelt verkettete Liste ohne Pointer definieren, mit Hilfe von Arrays
- Für DLLA besprechen wir eine andere Möglichkeit von Repräsentierung für verkettete Listen auf Arrays:
  - Die Grundidee ist gleich, man benutzt Indexe in dem Array für die Links zwischen den Elementen
  - Wir benutzten dieselbe Information, aber es wird anders strukturiert
  - Es ähnelt mehr der verketteten Listen mit dynamischer Allokation

# DLLA - Knoten

- Man kann eine Knotenstruktur auch mit Hilfe von Arrays definieren
- Ein Knoten (für eine doppelt verkettete Liste) enthält die Daten und die Links zu dem vorigen und nächsten Knoten:

DLLANode:

info: TElem

next: Integer

prev: Integer

# DLLA

- Um die Liste zu repräsentieren brauchen wir jetzt ein Array von *DLLANodes*
- Da es sich um eine doppelt verkettete Liste handelt, speichern wir den Head und den Tail der Liste

DLLA:

```
nodes: DLLANode[]
cap: Integer
head: Integer
tail: Integer
firstEmpty: Integer
size: Integer                    //nicht verpflichtend, aber manchmal nützlich
```

# DLLA – Speicherplatz allokieren und freigeben

- Damit die Repräsentierung und Implementierung ähnlich zu dynamisch allokierte verkettete Listen sind, kann man auch die Funktionen *allocate* und *free* definieren

```
function allocate(dlla) is:
//pre: dlla ist ein DLLA
//post: ein neues Element wird allokiert und die Position wird zurückgegeben
    newElem ← dlla.firstEmpty
    if newElem ≠ -1 then
        dlla.firstEmpty ← dlla.nodes[dlla.firstEmpty].next
        if dlla.firstEmpty ≠ -1 then
            dlla.nodes[dlla.firstEmpty].prev ← -1
        end-if
        dlla.nodes[newElem].next ← -1
        dlla.nodes[newElem].prev ← -1
    end-if
    allocate ← newElem
end-function
```

# DLLA – Speicherplatz allokieren und freigeben

```
subalgorithm free (dlla, pos) is:
//pre: dlla ist ein DLLA, pos ist eine ganze Zahl
//post: die Position pos wurde freigegeben
    dlla.nodes[pos].next ← dlla.firstEmpty
    dlla.nodes[pos].prev ← -1
     if dlla.firstEmpty ≠ -1 then
        dlla.nodes[dlla.firstEmpty].prev ← pos
    end-if
    dlla.firstEmpty ← pos
end-subalgorithm
```

# DLLA - insertPosition

```
subalgorithm insertPosition(dlla, elem, pos) is:
//pre: dlla ist ein DLLA, elem ist ein TElem, pos ist eine ganze Zahl
//post: das Element elem wird in dlla an der Position pos eingefügt
    if pos < 1 or pos > dlla.size +1 then
        @throw Exception
    end-if
    newElem ← allocate(dlla)
    if newElem = -1 then
        @resize
        newElem ← allocate(dlla)
    end-if
    dlla.nodes[newElem].info ← elem
    if pos = 1 then
        if dlla.head = -1 then
            dlla.head ← newElem
            dlla.tail ← newElem
        else
//Fortsetzung auf der nächsten Folie ...
```

# DLLA - insertPosition

```
            dlla.nodes[newElem].next ← dlla.head
            dlla.nodes[dlla.head].prev ← newElem
            dlla.head ← newElem
        end-if
    else
        nodC ← dlla.head
        posC ← 1
        while nodC ≠ -1 and posC < pos - 1 execute
            nodC ← dlla.nodes[nodC].next
            posC ← posC + 1
        end-while
        if nodC ≠ -1 then
            nodNext ← dlla.nodes[nodC].next
            dlla.nodes[newElem].next ← nodNext
            dlla.nodes[newElem].prev ← nodC
            dlla.nodes[nodC].next ← newElem
//Fortsetzung auf der nächsten Folie ...
```

# DLLA - insertPosition

```
            if nodNext = -1 then
                dlla.tail ← newElem
            else
                dlla.nodes[nodNext].prev ← newElem
            end-if
        end-if
    end-if
end-subalgorithm
```

Komplexität: $O(n)$

# DLLA - Iterator

- Der Iterator für DLLA enthält als *aktuelle Element* den Index für den aktuellen Knoten aus dem Array

DLLAIterator:
    list: DLLA
    currentElement: Integer

# DLLAIterator - init

**subalgorithm** init(it, dlla) **is**:

//pre: *dlla* ist ein DLLA

//post: *it* ist ein DLLAIterator für *dlla*

    it.list ← dlla

    it.currentElement ← dlla.head

**end-subalgorithm**

- Für ein (ADT) (dynamisches) Array wird *currentElement* mit 1 initialisiert bei dem Erstellen des Iterators.
- Für DLLA muss das *currentElement* zu dem Listenkopf zeigen (dieser kann sich auf Position 1 oder auf einer anderen Position befinden)

# DLLAIterator - getCurrent

```
subalgorithm getCurrent(it) is:
//pre: it ist ein DLLAIterator, it ist gültig
//post: e ist ein TElem, e ist das aktuelle Element aus it
//throws ein Exception falls it nicht gültig ist
        if it.currentElement = -1 then
            @throw Exception
        end-if
        getCurrent ← it.list.nodes[it.currentElement].info
end-subalgorithm
```

Komplexität: $\Theta(1)$

# DLLAIterator - next

```
subalgoritm next (it) is:
//pre: it ist ein DLLAIterator, it ist gültig
//post: das aktuelle Element aus it wird zu dem nächsten Element verschoben
//throws ein Exception falls it nicht gültig ist
        if it.currentElement = -1 then
            @throw Exception
        end-if
        it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm
```

- Für ein (dynamisches) Array wird *currentElement* mit 1 inkrementiert wenn man zu dem nächsten Element geht.
- Für DLLA muss man die Links folgen.
- Komplexität: $\Theta(1)$

# DLLAIterator - valid

```
function valid (it) is:
//pre: it ist ein DLLAIterator
//post: valid gibt den Wert wahr zurück falls das aktuelle Element gültig
// ist, falsch ansonsten
        if it.currentElement = -1 then
            valid ← False
        else
            valid ← True
        end-if
end-function
```

Komplexität: $\Theta(1)$

# Binärer Heap (auch Halde oder Haufen)

- Binäre Heaps sind eine einfache und effiziente Implementierung von Prioritätswarteschlangen (priority queues)

- Ein binärer Heap ist ein „Hybrid“ zwischen dynamisches Array und Binärbaum

- Ein binärer Heap ist ein Array, welches man als „fast vollständigen“ binären Baum mit besonderen Eigenschaften auffassen kann

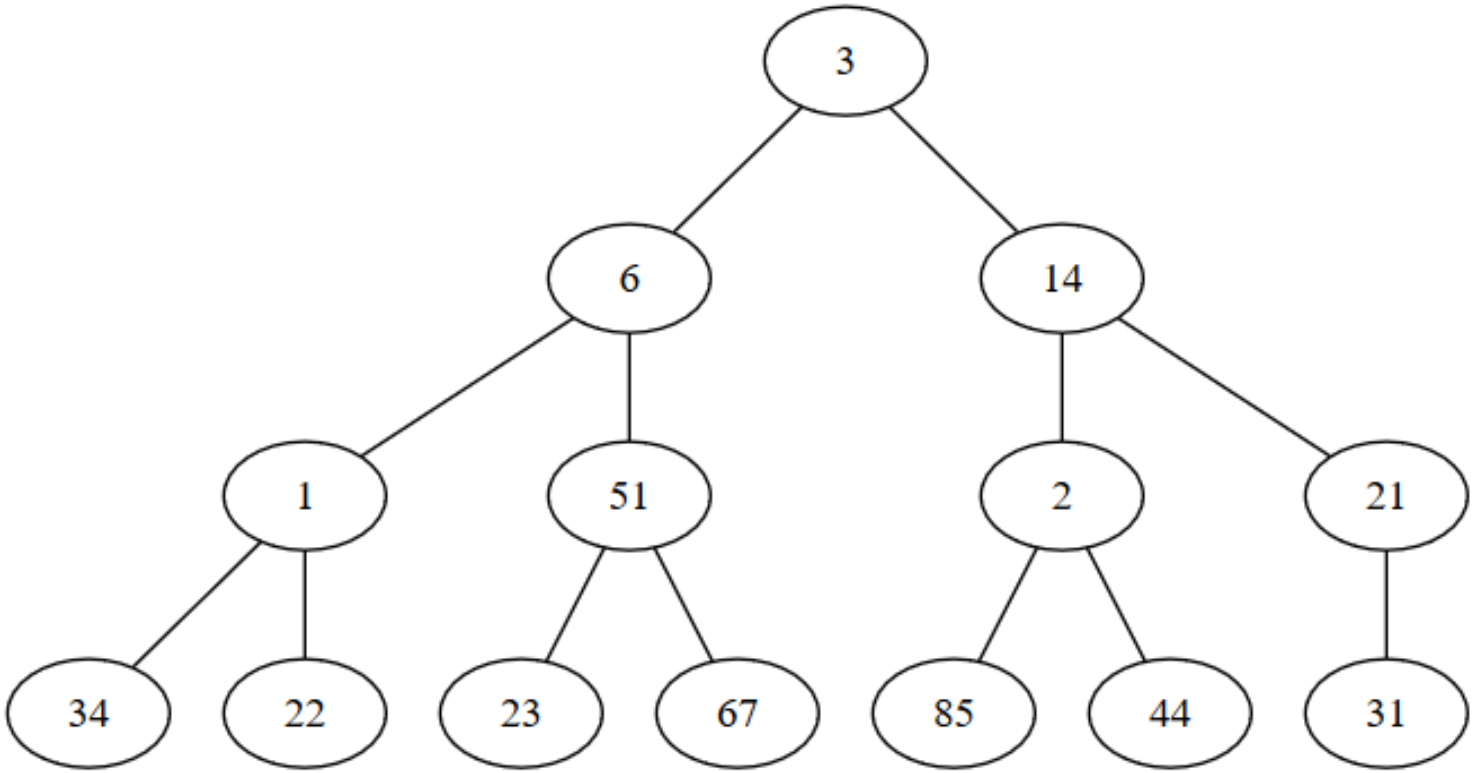
# Binärer Heap

- Nehmen wir an, dass folgendes Array auf der unteren Zeile Positionen und auf der oberen Zeile Elemente enthält:

| 3 | 6 | 14 | 1 | 51 | 2 | 21 | 34 | 22 | 23 | 67 | 85 | 44 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

- Man kann dieses Array als Binärbaum visualisieren, indem jeder Knoten genau zwei Kinder hat, außer der letzten zwei Niveaus, wo die Knoten von links nach rechts ausgefüllt werden

# Array visualisiert als Binärbaum




| 3 | 6 | 14 | 1 | 51 | 2 | 21 | 34 | 22 | 23 | 67 | 85 | 44 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

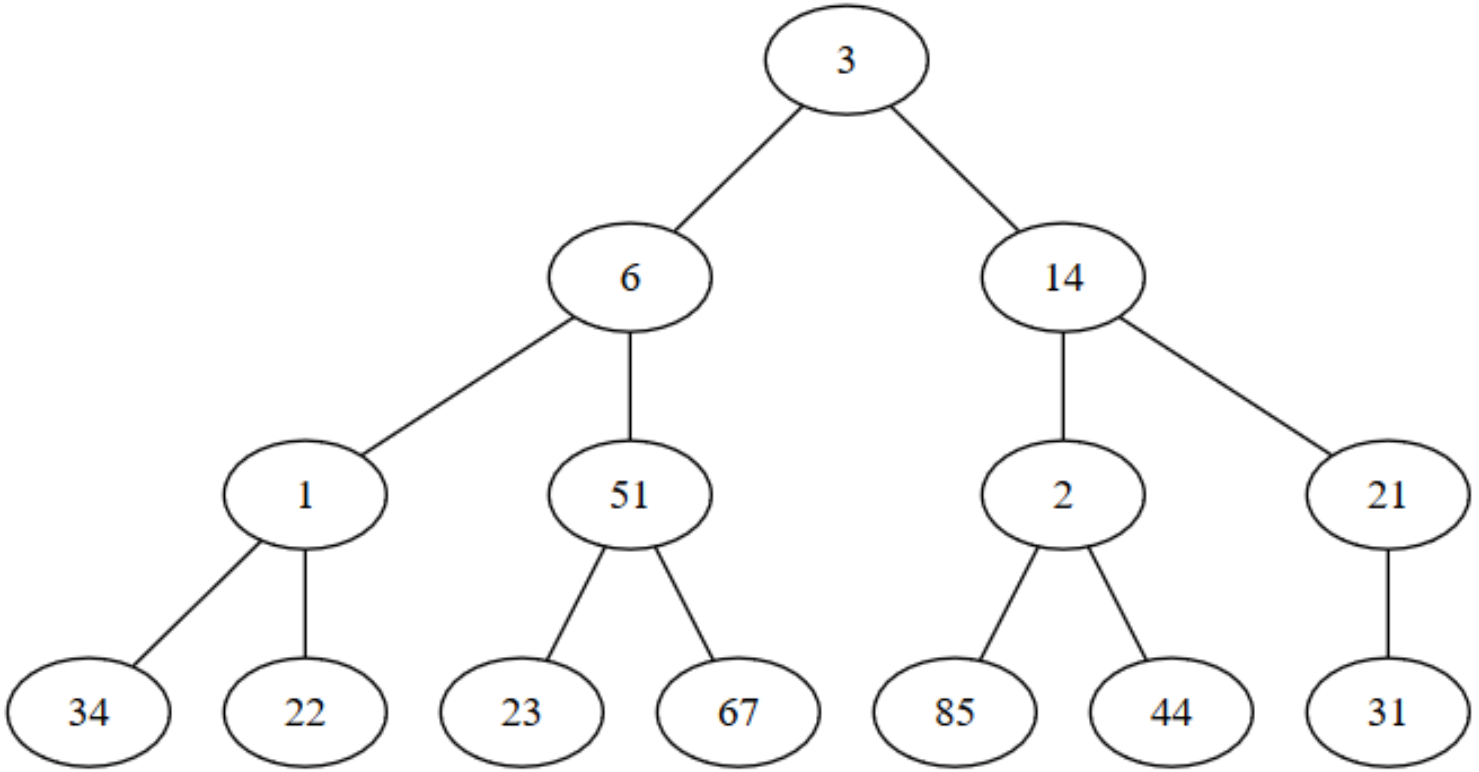
# Binärer Heap

- Wenn die Elemente des Arrays $a_1$, $a_2$, $a_3$, ... $a_n$ *sind,* dann gilt Folgendes:
  - $a_1$ stellt die Wurzel dar
  - Die Kinder der Wurzel entsprechen $a_2$ *und* $a_3$
  - Die Kinder von Knoten *i* haben Indizes 2 * *i* und 2 * *i* + 1 (falls diese Zahlen in 1..*n* liegen).
  - Der Vorgänger eines Knotens *i*, mit $2 \leq i \leq n$, hat den Index $\lfloor i/2 \rfloor$.

# Binärer Heap

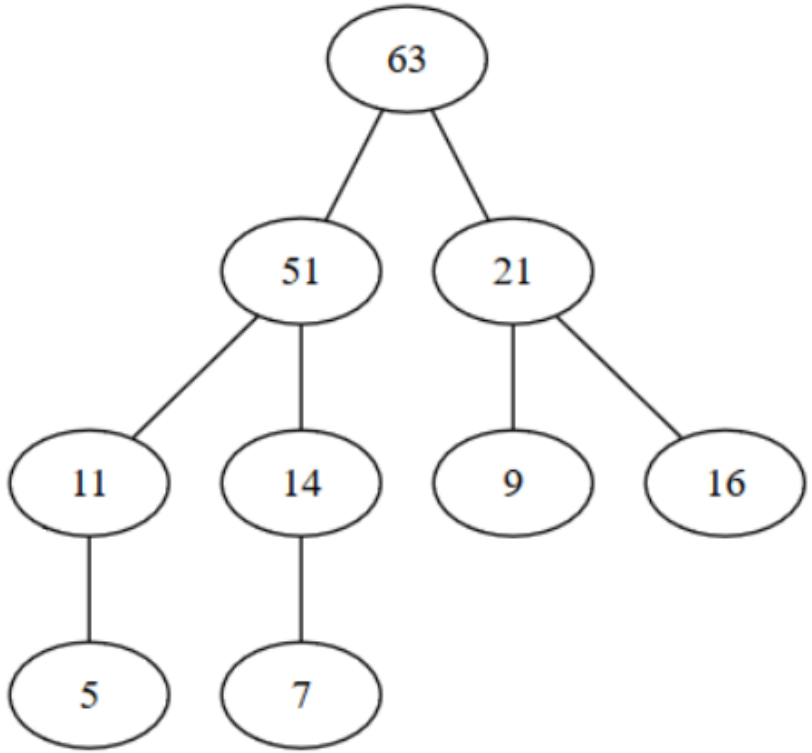
- Ein binären Heap **ist ein Array**, das als Binärbaum visualisiert werden kann und, dass zusätzlich die ***Heap-Struktur*** und ***Heap-Eigenschaft*** besitzt
  - Heap-Struktur: ein Binärbaum, in welchem jeder Knoten genau zwei Kinder hat, außer der letzten zwei Niveaus, wo die Knoten von links nach rechts ausgefüllt werden
  - Heap-Eigenschaft: $a_i \geq a_{2*i}$ (falls $2 * i \leq n$) und $a_i \geq a_{2*i+1}$ (falls $2 * i + 1 \leq n$)
  - Ein Baum erfüllt die Heap-Eigenschaft bezüglich einer Vergleichsrelation „$\geq$“ auf den Schlüsselwerten genau dann, wenn für jeden Knoten *u* des Baums gilt, dass *u.wert* $\geq$ *v.wert* für alle Knoten *v* aus den Unterbäumen von *u* (**man kann jedwelche Vergleichsrelation auswählen**)

# Ist das ein Heap?



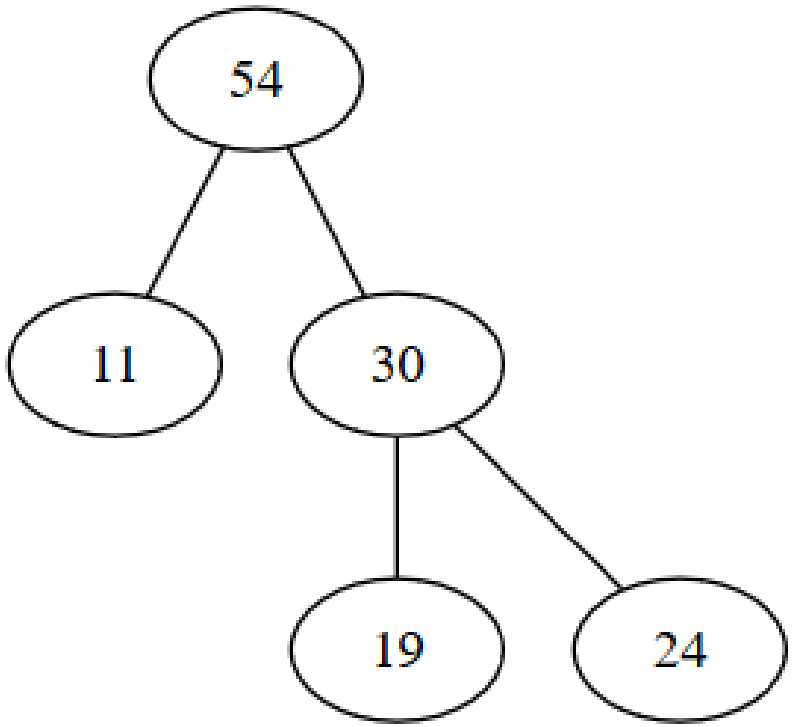

- **Nein**. Es hat die Heap-Struktur, aber nicht die Heap-Eigenschaft!

# Ist das ein Heap?

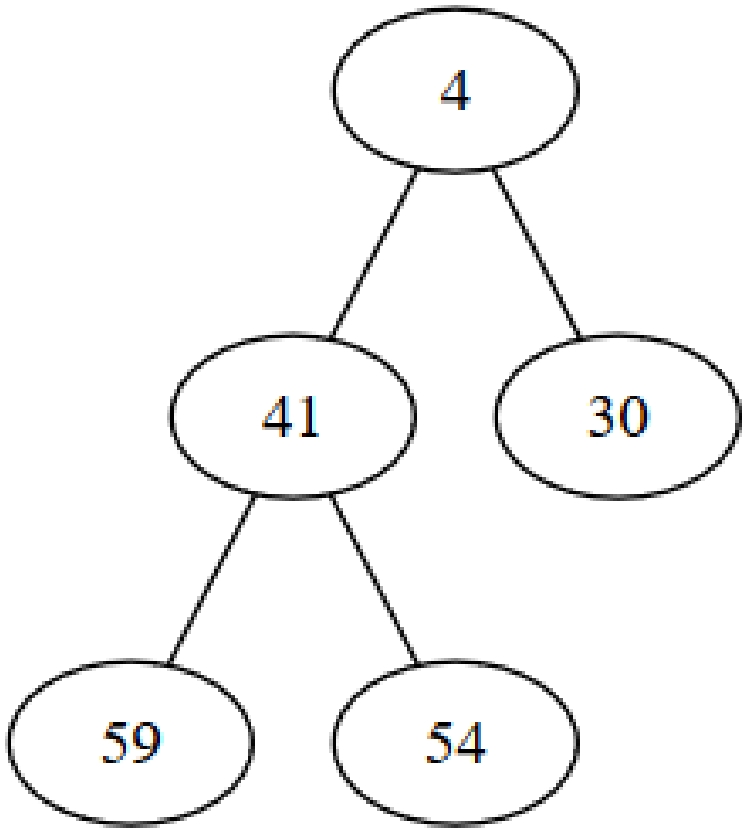



- **Nein**. Es hat die Heap-Eigenschaft, aber nicht die Heap-Struktur!

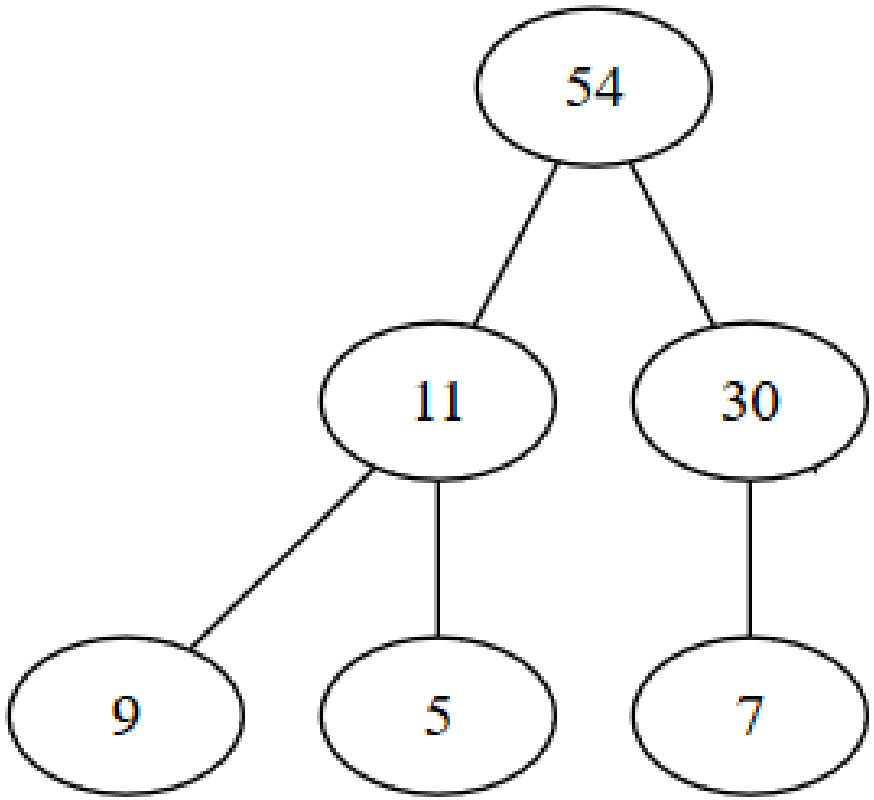
# Ist das ein Heap?



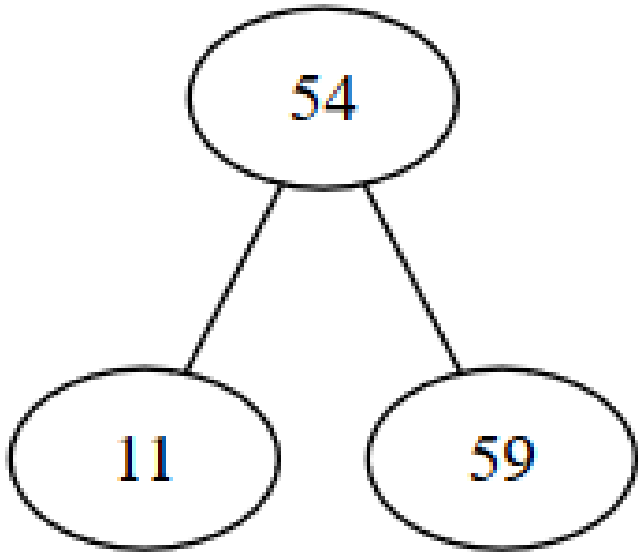
# Ist das ein Heap?



# Ist das ein Heap?



# Ist das ein Heap?



# Binärer Heap

- Sind folgende Arrays gültige Heaps? Falls die Antwort negativ ist, dann wandle diese in gültige Heaps um durch das Vertauschen zweier Elemente.
  a) [70, 10, 50, 7, 1, 33, 3, 8]
  b) [1, 2, 4, 8, 16, 32, 64, 65, 10]
  c) [10, 12, 104, 60, 13, 102, 101, 80, 90, 14, 15, 16]

# Binärer Heap

- In Abhängigkeit von der gewählten Vergleichsrelation erhält man nun einen Baum, der entweder das Minimum oder das Maximum in der Wurzel enthält:
  - Ein binärer Baum, der die Heap-Eigenschaft mit der Relation „$\geq$“ erfüllt, wird als **Max-Heap** bezeichnet.
  - Ein binärer Baum, der die Heap-Eigenschaft mit der Relation „$\leq$“ erfüllt, wird als **Min-Heap** bezeichnet.
- Der Baum, der ein binärer Heap mit Größe *n* repräsentiert, besitzt eine Höhe von $log_2 n$

# Heap - Operationen

- Ein Heap kann als Repräsentierung für eine Prioritätswarteschlange benutzt werden und enthält zwei spezifische Operationen:
  - Füge ein neues Element in einen Heap ein (so dass man die Struktur des Heaps und die Heap-Eigenschaft behaltet)
  - Lösche ein Element – **man löscht immer nur die Wurzel des Heaps** und kein anderes Element

# Heap - Repräsentierung mit dynamischem Array

Heap:

cap: Integer

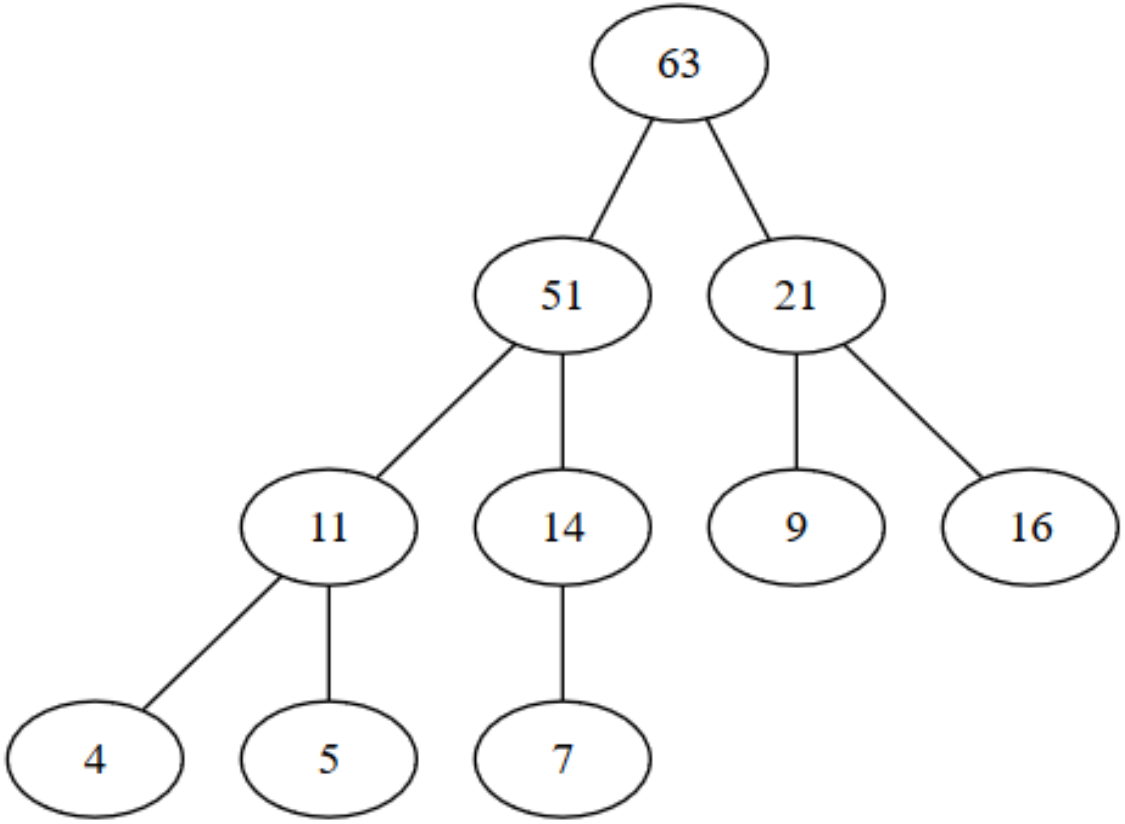
len: Integer

elems: TElem[]

- Für die Implementierung nehmen wir an, dass es um eine Max-Heap geht
- Bei Bedarf kann man eine allgemeine, abstrakte Relation definieren

# Binärer Heap – add

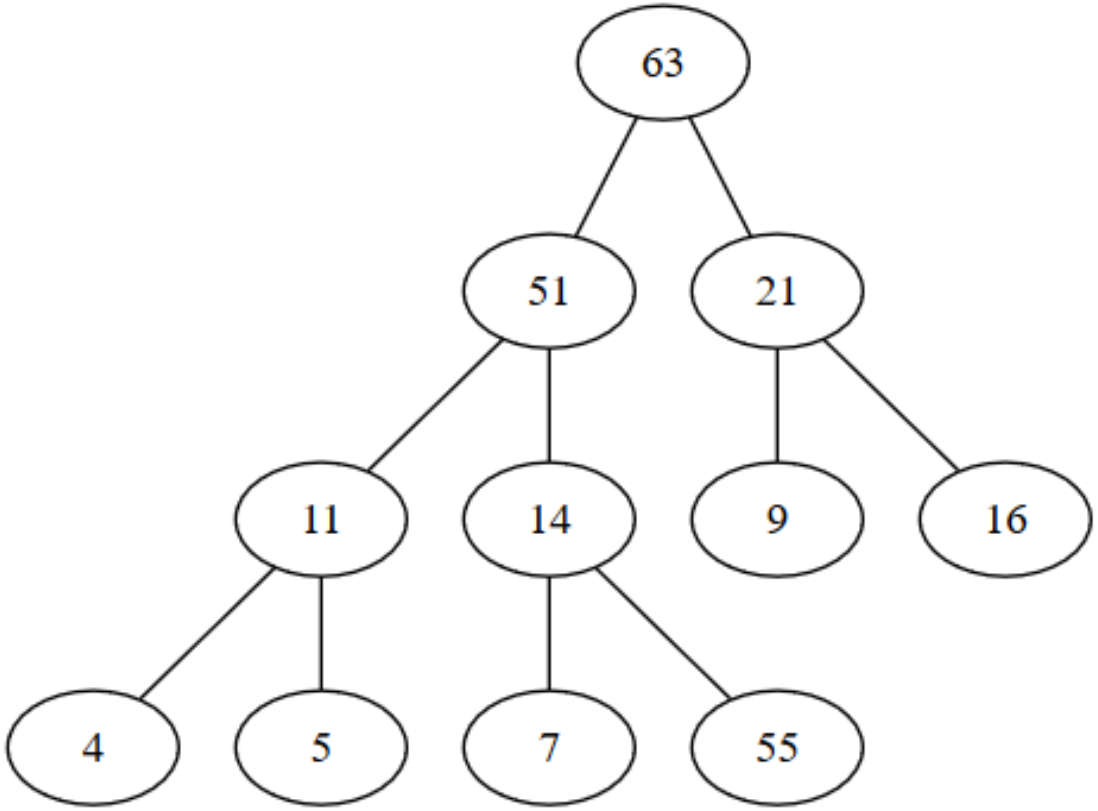
- Sei der folgende Max-Heap:




- Füge das Element 55 in den Heap ein

# Binärer Heap – add

- Um die Heap-Struktur zu behalten, fügen wir den neuen Knoten als das rechte Kind von dem Knoten mit Wert 14 ein (am Ende des Arrays)



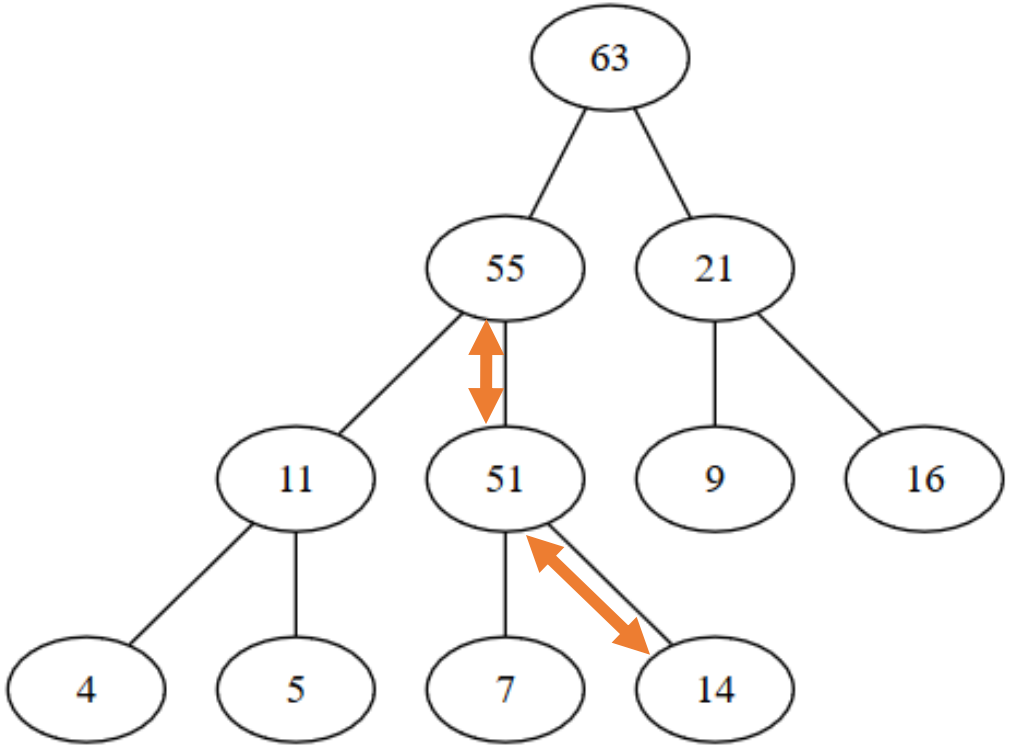
# Binärer Heap – add

- Die Heap-Eigenschaft ist nicht erfüllt: 14 hat als Kind ein Knoten mit Wert 55 (in einem Max-Heap sollte jeder Knoten größer als seine Kinder sein)

- Um die Heap-Eigenschaft zu bewahren, beginnt man die Einträge der Knoten zu vertauschen:
  - Man lässt den neuen Knoten durch sukzessives Vertauschen mit seinem Vaterknoten so weit im Baum „hochsteigen“, bis die Heap-Eigenschaft wiederhergestellt ist (*bubble-up*)

# Binärer Heap – add

- Nach dem *bubble-up:*



# Binärer Heap – add

```
subalgorithm add(heap, e) is:
//heap - ein Heap
//e - das Element, das eingefügt werden muss
        if heap.len = heap.cap then
            @ resize
        end-if
        heap.elems[heap.len+1] ← e
        heap.len ← heap.len + 1
        bubble-up(heap, heap.len)
end-subalgorithm
```

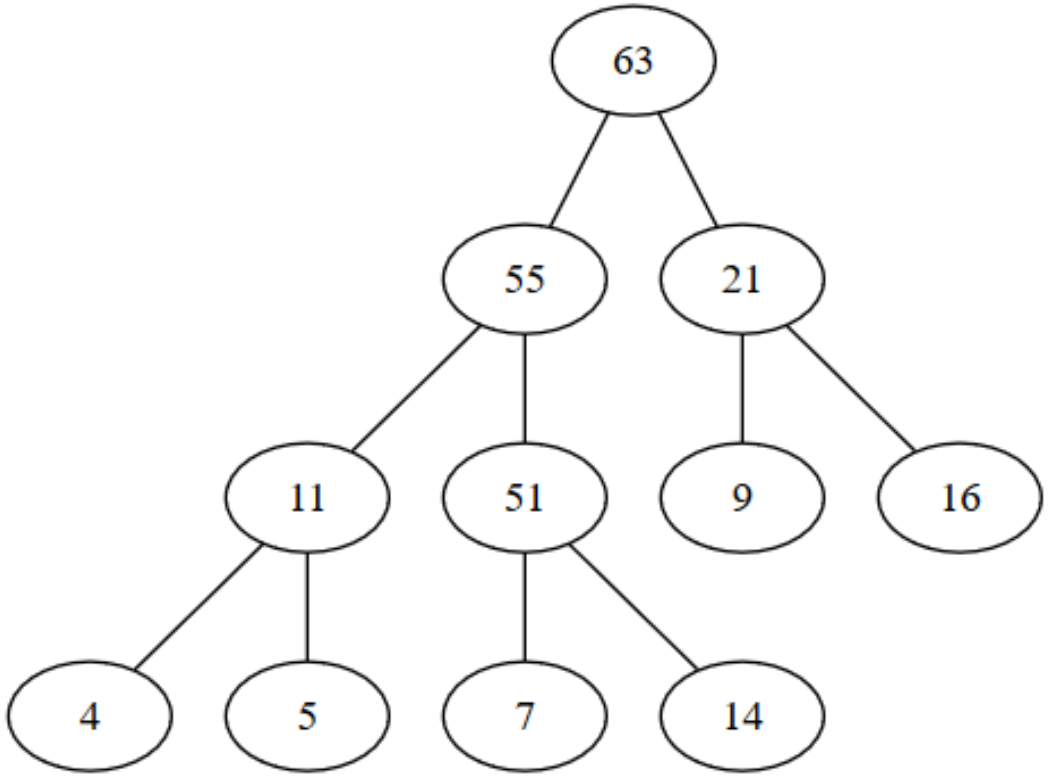
# Binärer Heap – add

```
subalgorithm bubble-up (heap, p) is:
//heap - ein Heap
//p - Position von der man den neuen Knoten hochsteigen muss
    poz ← p
    elem ← heap.elems[p]
    parent ← p / 2
    while poz > 1 and elem > heap.elems[parent] execute
        //der Vaterknoten wird nach unten verschoben
        heap.elems[poz] ← heap.elems[parent]
        poz ← parent
        parent ← poz / 2
    end-while
    heap.elems[poz] ← elem  //wir haben die richtige Position des Elementes gefunden
end-subalgorithm
```

- Komplexität: $O(log_2 n)$ - da wir pro Ebene des Baumes nur konstanten Aufwand investieren und der Baum logarithmische Höhe besitzt
- Gibt es einen besten Fall, in dem die Komplexität besser ist als $log_2 n$ ?

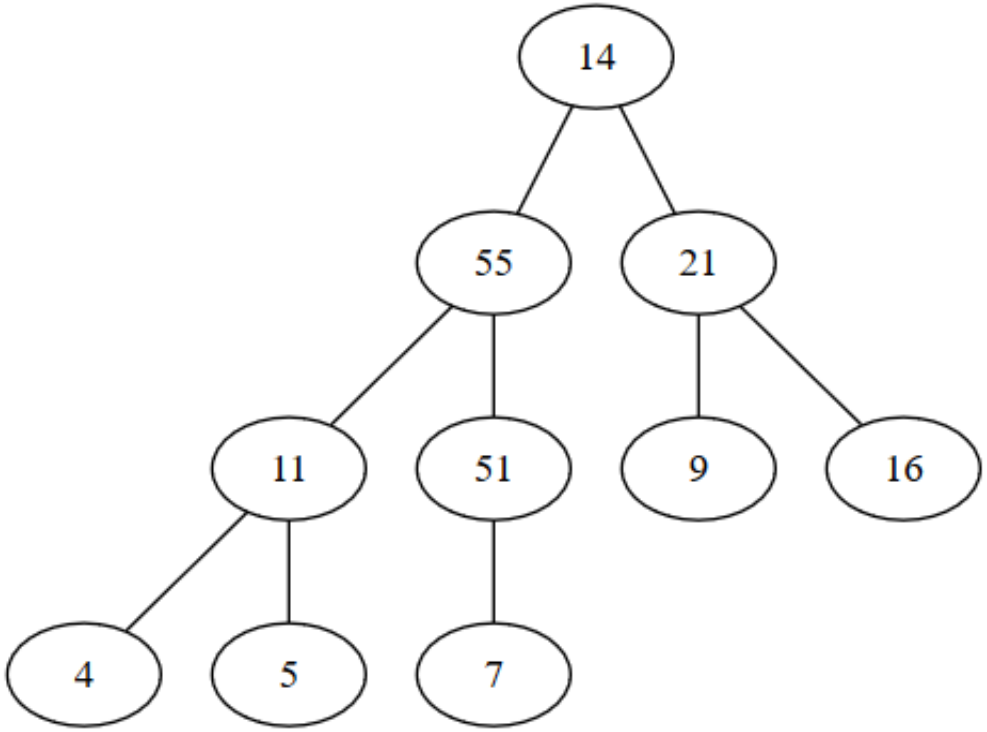
# Binärer Heap – remove

- Aus einem Heap kann man nur die Wurzel löschen



# Binärer Heap – remove

- Um die Heap-Struktur zu behalten, ersetzt man die Wurzel durch das letzte Element aus dem Array



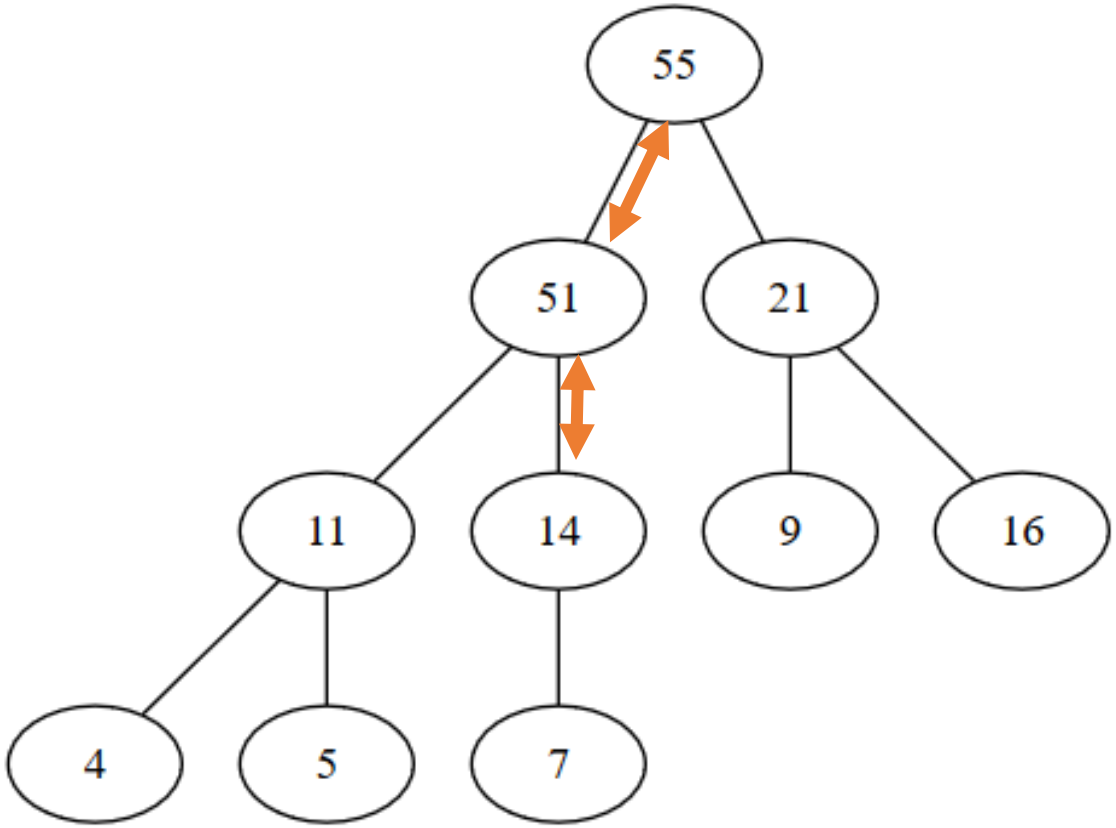
# Binärer Heap – remove

- Die Heap-Eigenschaft ist nicht erfüllt: die Wurzel ist nicht mehr das größte Element

- Nun lassen wir die neue Wurzel im Heap durch Vertauschen mit dem größten Kind so weit in den Heap „absinken“, bis die Heap-Eigenschaft wieder erfüllt ist (bis der Knoten größer als beide Kinder ist oder ein Blatt ist) (*bubble-down*)

# Binärer Heap – remove

- Nach dem *bubble-down:*



# Binärer Heap – remove

```
function remove(heap) is:
//heap - ist ein Heap
    if heap.len = 0 then
        @ error - leeren Heap
    end-if
    deletedElem ← heap.elems[1]
    heap.elems[1] ← heap.elems[heap.len]
    heap.len ← heap.len - 1
    bubble-down(heap, 1)
    remove ← deletedElem
end-function
```

# Binärer Heap – remove

```
subalgorithm bubble-down(heap, p) is:
//heap - ist ein Heap
//p - Position von der man den neuen Knoten absinken muss
    poz ← p
    elem ← heap.elems[p]
    while poz < heap.len execute
        maxChild ← -1
        if poz * 2 ≤ heap.len then
        //es gibt ein linkes Kind
            maxChild ← poz*2
        end-if
        if poz*2+1 ≤ heap.len and heap.elems[2*poz+1] > heap.elems[2*poz] then
        //der Knoten hat zwei Kinder und das rechte ist größer
            maxChild ← poz*2 + 1
        end-if
//Fortsetzung auf der nächsten Folie
```

# Binärer Heap – remove

```
            if maxChild ≠ -1 and heap.elems[maxChild] > elem then
                tmp ← heap.elems[poz]
                heap.elems[poz] ← heap.elems[maxChild]
                heap.elems[maxChild] ← tmp
                poz ← maxChild
            else
                poz ← heap.len + 1
                //um die while-Schleife zu stoppen
             end-if
        end-while
    end-subalgorithm
```

- Komplexität: $O(log_2 n)$

# Übungen

- Fange mit einem leeren Max-Heap an und füge, in der gegebenen Reihenfolge, folgende Werte ein: 8, 27, 13, 15*, 32, 20, 12, 50*, 29, 11*. Zeichne den Heap neu nachdem die Elemente markiert mit * eingefügt werden. Lösche 3 Elemente aus dem Heap und zeichne den Heap neu nach jeder Löschoperation.

- Fange mit einem leeren Min-Heap an und füge, in der gegebenen Reihenfolge, folgende Werte ein: 15, 17, 9, 11, 5, 19, 7. Lösche alle Elemente aus dem Heap der Reihe nach. Zeichne den Heap neu nach jeder zweiten Löschoperation.

# Denk darüber nach

- Wo kann man in einem Max-Heap:
  - Das größte Element des Arrays finden?
  - Das kleinste Element des Arrays finden?