Datenstrukturen

Mariano Zelke

Sommersemester 2012

Tiefensuche: Die globale Struktur

- Der gerichtete oder ungerichtete Graph G werde durch seine Adjazenzliste A repräsentiert.
- Im Array besucht wird vermerkt, welche Knoten bereits besucht wurden.

```
void Tiefensuche(){
  for (int k = 0; k < n; k++) besucht[k] = 0;
  for (int k = 0; k < n; k++)
     if (! besucht[k])
        tsuche(k);
}
```

- Jeder Knoten wird besucht,
- aber tsuche(v) wird nur dann aufgerufen, wenn v nicht als „besucht“ markiert ist.

tsuche()

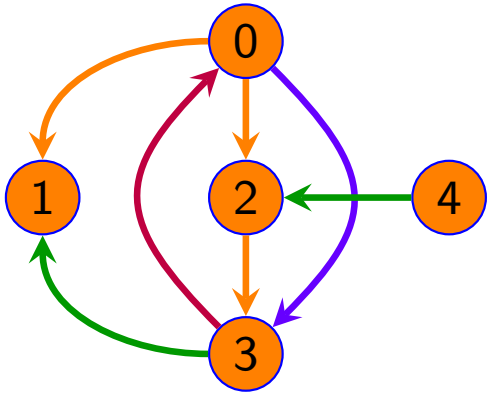
Die Knoten des Graphen werden definiert durch

```
struct Knoten {
    int name;
    Knoten *next; }
```

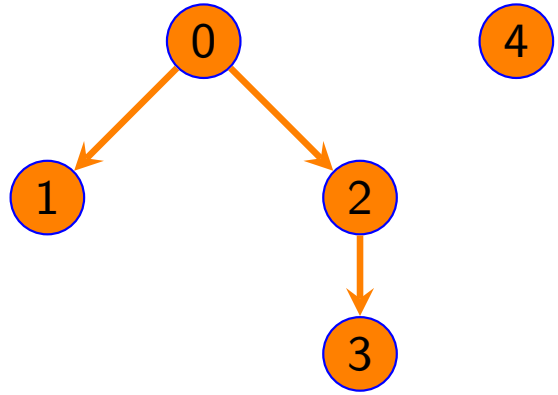
- Als erste Aktion von tsuche(v) wird v als besucht markiert.
- Dann wird tsuche(v) für alle unmarkierten Nachbarn/Nachfolger von v rekursiv aufgerufen.

```
void tsuche(int v){
   besucht[v] = 1;
   Knoten *p;
   for (p = A[v]; p !=0; p = p->next)
      if (!besucht [p->name])
         tsuche(p->name);
}
```

Tiefensuche für gerichtete Graphen I



Wald der Tiefensuche:



Angenommen, die Tiefensuche startet im Knoten 0 und in jeder Adjazenzliste sind die Knoten aufsteigend sortiert. Dann erhalten wir vier verschiedene Kantentypen:

- Baumkanten: $(0, 1)$, $(0, 2)$ und $(2, 3)$, sie bilden den Wald W_G der Tiefensuche
- Rückwärtskante: $(3, 0)$, sie verbindet einen Knoten mit seinem Vorgänger in W_G
- Querkanten: $(3, 1)$ und $(4, 2)$, sie verbinden zwei Knoten, die in W_G nicht miteinander in einer Nachfolger-Vorgänger-Beziehung stehen.
- Vorwärtskante: $(0, 3)$, sie verbindet einen Knoten mit einem Nachfahren in W_G, der kein Kind ist.

Tiefensuche für gerichtete Graphen II

Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste vorliegt.

(a) Tiefensuche besucht jeden Knoten genau einmal.

(b) Die Laufzeit von Tiefensuche() ist durch $\mathcal{O}(|V| + |E|)$ beschränkt.

(c) tsuche(v) besucht einen Knoten w genau dann, wenn w auf einem Weg liegt, dessen Knoten vor Beginn von tsuche(v) unmarkiert sind.

- Zu Beginn von tsuche(v) sei w von v aus durch einen „unmarkierten Weg“ erreichbar.
- Damit kann tsuche(v) nur dann terminieren, wenn w während der Ausführung von tsuche(v) markiert wird.
- Also wird w besucht.

Die folgende Aussage ist falsch: tsuche(v) wird genau die Knoten besuchen, die auf einem Weg mit Anfangsknoten v liegen.

Kantentypen für gerichtete Graphen

- Es ist nicht verwunderlich, dass durch die Kantenrichtungen neben Rückwärtskanten jetzt auch Vorwärtskanten vorkommen.
- Querkanten sind ein gänzlich neuer Kantentyp. Ein Querkante heißt eine rechts-nach-links Querkante, wenn sie von einem später besuchten Knoten zu einem früher besuchten Knoten führt.

Es gibt nur rechts-nach-links Querkanten.

Warum? Sei $e = (v, w)$ eine beliebige Kante.

- Wenn tsuche(v) terminiert, dann ist w markiert.
- Wenn w vor dem Aufruf von tsuche(v) markiert wurde: e ist entweder eine Rückwärtskante oder eine rechts-nach-links Querkante.
- Wenn w während des Aufrufs markiert wird, dann ist e eine Baumkante oder eine Vorwärtskante.

Eine automatische Erkennung der Kantentypen

Wir benutzen zwei integer-Arrays „Anfang“ und „Ende“ als Uhren, um den Zeitpunkt des Beginns und des Endes eines Besuchs festzuhalten.

```
Anfangnr=Endenr=0;
void tsuche(int v){
   Anfang[v] = ++Anfangnr;
   Knoten *p;
   for (p = A[v]; p != 0; p = p->next)
      if (!Anfang[p->name])          //p->name ist unbesucht
         tsuche(p->name);
   Ende[v] = ++Endenr;
}
```

- $e = (u, v)$ ist eine Vorwärtskante
  $\Leftrightarrow$ Anfang$[u] <$ Anfang$[v]$ und $e = (u, v)$ ist keine Baumkante.
- $e = (u, v)$ ist eine Rückwärtskante
  $\Leftrightarrow$ Anfang$[u] >$ Anfang$[v]$ und Ende$[u] <$ Ende$[v]$.
- $e = (u, v)$ ist eine Querkante
  $\Leftrightarrow$ Anfang$[u] >$ Anfang$[v]$ und Ende$[u] >$ Ende$[v]$.

Anwendungen der Tiefensuche I

Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste repräsentiert ist. Dann lassen sich die folgenden Probleme in Zeit $\mathcal{O}(|V| + |E|)$ lösen:

(a) Ist G azyklisch?

- Jede Rückwärtskante schließt einen Kreis.
- Baum-, Vorwärts- und Querkanten allein können keinen Kreis schließen.
- G ist azyklisch genau dann, wenn G keine Rückwärtskanten hat.

(b) Führe eine topologische Sortierung durch.

- Führe eine Tiefensuche durch.
- G muss azyklisch sein, hat also nur Baum-, Vorwärts- und rechts-nach-links Querkanten.
- „Sortiere“ die Knoten absteigend nach ihrem Endewert: keine Kante führt von einem Knoten mit kleinerem Endewert zu einem Knoten mit großem Endewert.

Anwendungen der Tiefensuche II

G ist stark zusammenhängend, wenn es für jedes Knotenpaar (u, v) einen Weg von u nach v gibt.

(c) Ist G stark zusammenhängend? Es genügt zu zeigen, dass alle Knoten von Knoten 0 aus erreichbar sind und dass jeder Knoten auch Knoten 0 erreicht.

- Alle Knoten sind genau dann von Knoten 0 aus erreichbar, wenn tsuche(0) alle Knoten besucht.
- Kann jeder Knoten den Knoten 0 erreichen?
  - Kehre die Richtung aller Kanten um,
  - führe tsuche(0) auf dem neuen Graphen aus
  - und überprüfe, ob alle Knoten besucht werden.

Wie bestimmt man kürzeste Wege? Tiefensuche ist völlig ungeeignet!

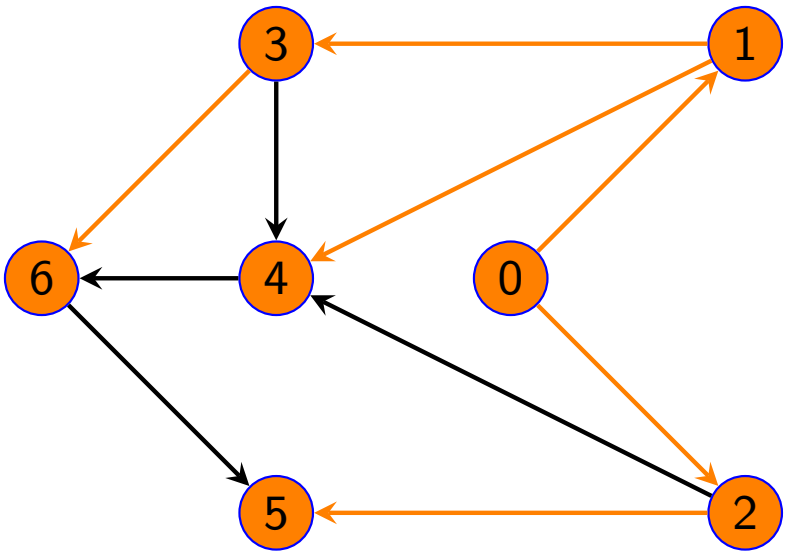
Breitensuche

Breitensuche für einen Knoten v soll zuerst v, dann die „Kindergeneration“ von v, dann die „Enkelkinder“ von v, dann die „Urenkel“ von v usw. besuchen.

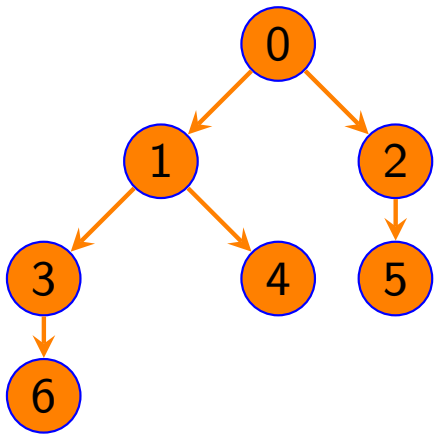
```
void Breitensuche(int v){
   Knoten *p; int w; queue q;

   for (int k =0; k < n ; k++) besucht[k] = 0;

   q.enqueue(v); besucht[v] = 1;

   while (!q.empty ( )){
     w = q.dequeue ( );
     for (p = A[w]; p != 0; p = p->next)
       if (!besucht[p->name]){
         q.enqueue(p->name);
         besucht[p->name] = 1;  //(w,p->name) ist Baumkante
       }
   }
}
```

Beispiel

Breitensuche(v) berechnet einen Baum mit Wurzel v, wenn wir alle Baumkanten einsetzen.



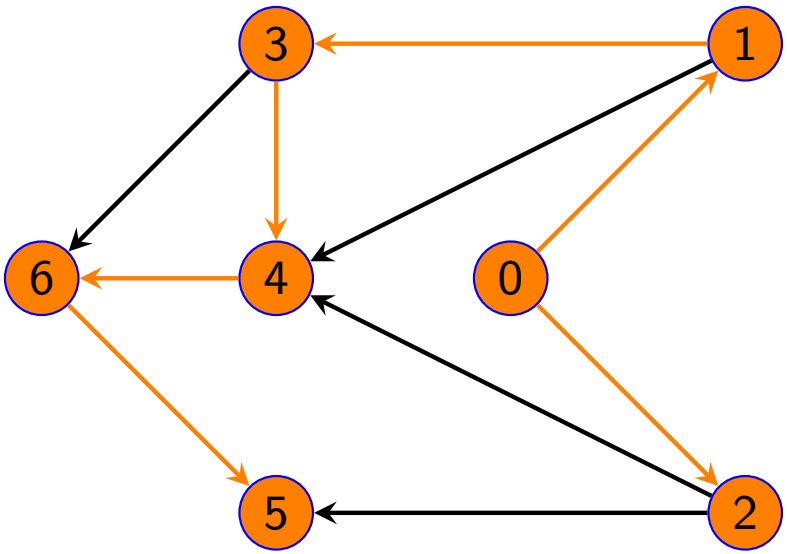
Baum der Breitensuche:



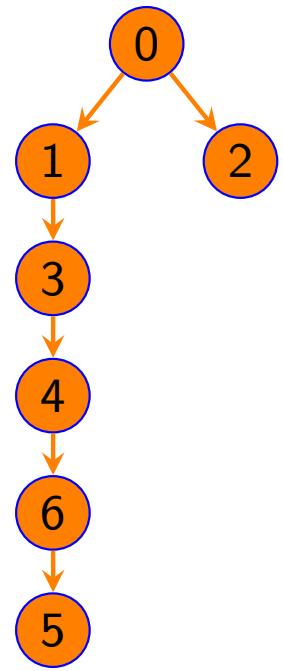
Wir beginnen sowohl Breitensuche wie auch Tiefensuche im Knoten 0. Wie sehen der Baum der Breitensuche und der Baum der Tiefensuche aus?

Beispiel

Breitensuche(v) berechnet einen Baum mit Wurzel v, wenn wir alle Baumkanten einsetzen.



Baum der Tiefensuche:

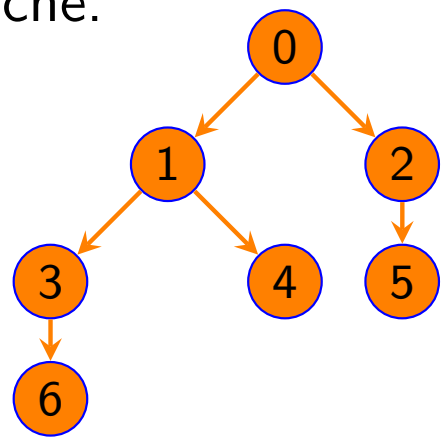


Wir beginnen sowohl Breitensuche wie auch Tiefensuche im Knoten 0. Wie sehen der Baum der Breitensuche und der Baum der Tiefensuche aus?

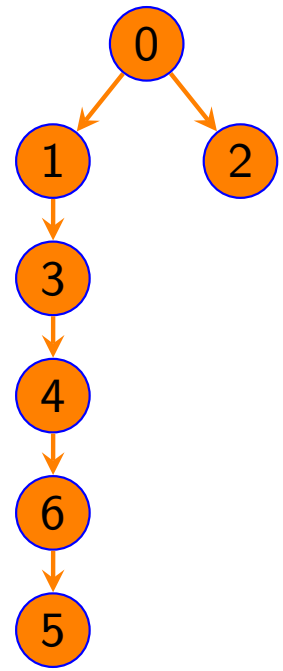
Beispiel

Breitensuche(v) berechnet einen Baum mit Wurzel v, wenn wir alle Baumkanten einsetzen.

Baum der Breitensuche:



Baum der Tiefensuche:



Wir beginnen sowohl Breitensuche wie auch Tiefensuche im Knoten 0. Wie sehen der Baum der Breitensuche und der Baum der Tiefensuche aus?
Und ein weiteres Beispiel hier.

Eigenschaften der Breitensuche I

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Für Knoten w setze $V_w = \{u \in V \mid \text{Es gibt einen Weg von } w \text{ nach } u\}$ und $E_w = \{e \in E \mid \text{beide Endpunkte von } e \text{ gehören zu } V_w\}$.

(a) Breitensuche(w) besucht jeden Knoten in V_w genau einmal und sonst keinen anderen Knoten.

- Nur bisher nicht besuchte Knoten werden in die Schlange eingefügt, dann aber sofort als besucht markiert: Jeder Knoten wird höchstens einmal besucht.
- Bevor Breitensuche(w) terminiert, müssen alle von w aus erreichbaren Knoten besucht werden.

(b) Breitensuche(w) benötigt Zeit höchstens $\mathcal{O}(|V_w| + |E_w|)$.

- Die Schlange benötigt Zeit höchstens $\mathcal{O}(|V_w|)$, da genau die Knoten aus V_w eingefügt werden und zwar genau einmal.
- Jede Kante wird für jeden Endpunkt genau einmal in seiner Adjazenzliste „angefasst“. Insgesamt Zeit $\mathcal{O}(|E_w|)$.

Eigenschaften der Breitensuche II

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Ein Baum $T(w)$ mit Wurzel w heißt ein Baum kürzester Wege, falls

- $T(w)$ die Knotenmenge V_w hat und
- falls für jeden Knoten $u \in V_w$, der Weg in $T(w)$ von der Wurzel w nach u ein kürzester Weg in G ist.

Im Baum B der Breitensuche:

- Zuerst werden Knoten u im Abstand 1 von w in die Schlange eingefügt: Die Kinder von w in B stimmen mit den Nachbarn von w in G überein.
- Zu jedem Zeitpunkt: Wenn ein Knoten u im Abstand d von der Wurzel aus der Schlange entfernt wird, dann werden höchstens Nachbarn von u im Abstand $d + 1$ von w eingefügt.
- Breitensuche baut seinen Baum „Generation für Generation“ auf.

Der Baum der Breitensuche ist ein Baum kürzester Wege.

Breitensuche und kürzeste Wege

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph, der als Adjazenzliste repräsentiert sei. Dann können wir in Zeit $\mathcal{O}(|V| + |E|)$ kürzeste Wege von einem Knoten w zu allen anderen Knoten bestimmen.

- Breitensuche(w) terminiert in Zeit $\mathcal{O}(|V| + |E|)$.
- Der Baum von Breitensuche ist ein Baum kürzester Wege!
- Wir können somit sämtliche kürzesten Wege kompakt als einen Baum darstellen:
  - Implementiere den Baum als Vater-Array.
  - Wir erhalten für jeden Knoten u einen kürzesten Weg von w nach u durch Hochklettern im Vater-Array.

Prioritätswarteschlangen

Der abstrakte Datentyp „Prioritätswarteschlange“: Füge Elemente (mit Prioritäten) ein und entferne jeweils das Element höchster Priorität.

- Eine Schlange ist eine sehr spezielle Prioritätswarteschlange: Die Priorität eines Elements ist der negative Zeitpunkt der Einfügung.
- Der abstrakte Datentyp „Prioritätswarteschlange“ umfasst dann die Operationen
  - void insert(x,Priorität),
  - int delete_max(),
  - void change_priority(wo,Priorität*), wähle Priorität* als neue Priorität
  - und void remove(wo), entferne das durch wo beschriebene Element.

Wir müssen eine geeignete Datenstruktur entwerfen.

Der Heap

Ein Heap ist ein Binärbaum mit Heap-Struktur, der Prioritäten gemäß einer Heap-Ordnung abspeichert.

Ein geordneter Binärbaum T der Tiefe t hat Heap-Struktur, wenn:

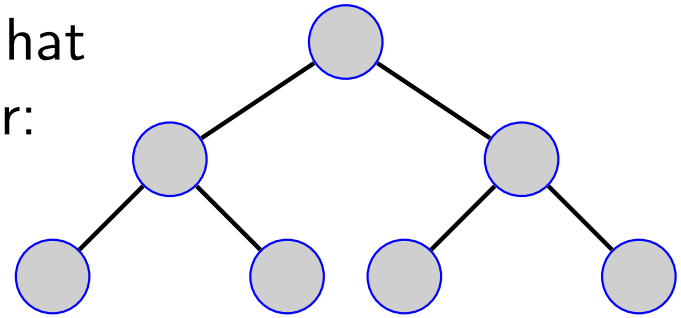
(a) jeder Knoten der Tiefe höchstens $t-2$ genau 2 Kinder hat,

(b) wenn ein Knoten v der Tiefe $t-1$ weniger als 2 Kinder hat, dann
- haben alle Knoten der Tiefe $t-1$, die rechts von v liegen, kein Kind
- wenn v genau ein Kind hat, dann ist es ein linkes Kind.
  (Daraus folgt, dass nur höchstens ein Knoten genau ein Kind haben kann.)

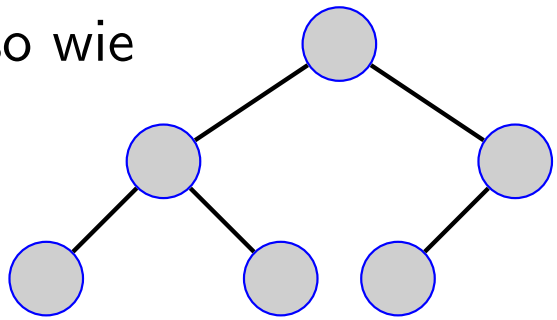
Ein Binärbaum mit Heap-Struktur ist ein fast vollständiger binärer Baum: Alle Knoten links von v haben zwei Kinder, alle Knoten rechts von v haben keine Kinder.

Beispiele Heap-Struktur

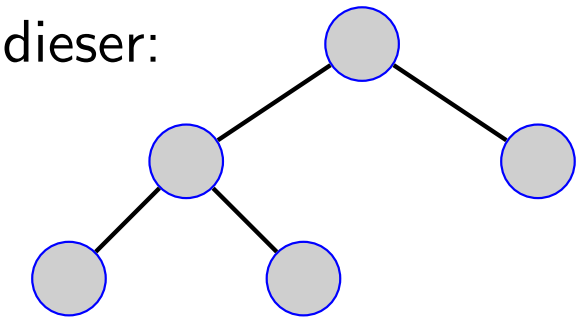
Dieser Baum hat Heap-Struktur:



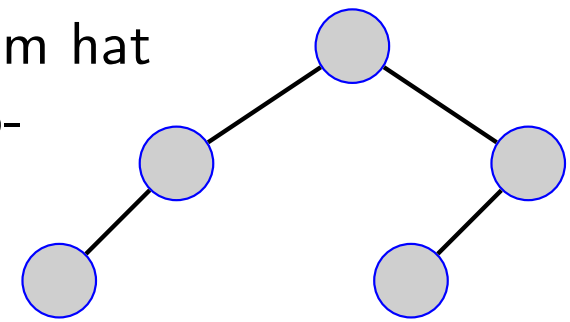
genauso wie dieser:



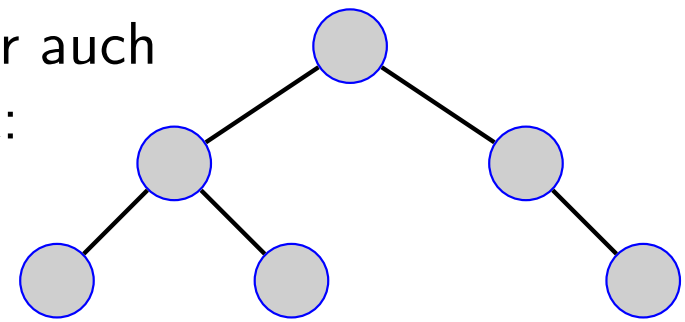
und dieser:



Dieser Baum hat *keine* Heap-Struktur:



dieser auch nicht:



Heap-Ordnung

Ein geordneter binärer Baum T mit Heap-Struktur speichere für jeden Knoten v die Priorität $p(v)$ von v.
T hat Heap-Ordnung, falls für jeden Knoten v und für jedes Kind w von v gilt

$$p(v) \geq p(w)$$

- Die höchste Priorität wird stets an der Wurzel gespeichert.

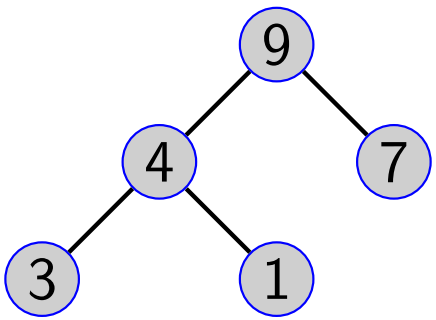
  Für die Operation delete_max() muss nur die Priorität der Wurzel überschrieben werden.
- Wie sollte man einen Baum mit Heap-Struktur implementieren?
  Wir arbeiten mit einem Array.

Die Datenstruktur Heap (Link)

Das Array H ist ein Heap für T, wenn T Heap-Struktur und Heap-Ordnung hat. Zusätzlich muss gelten

- $H[1] = p(r)$ für die Wurzel r von T und
- wenn $H[i]$ die Priorität des Knotens v speichert, dann gilt $H[2 \cdot i] = p(v_L)$ für das linke Kind v_L von v und $H[2 \cdot i + 1] = p(v_R)$ für das rechte Kind v_R.

Beispiel

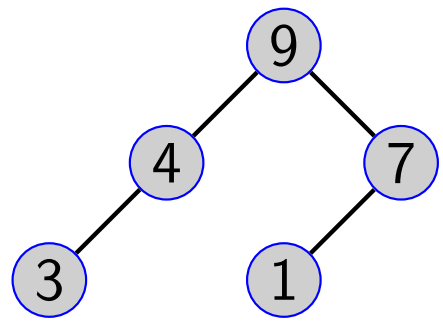


besitzt den Heap: (9,4,7,3,1)

dargestellt als Array

0	9	4	7	3	1

Der folgende Baum verletzt die Heap-Struktur:



Sein „Heap“ (9,4,7,3, ,1) enthält ein Loch.

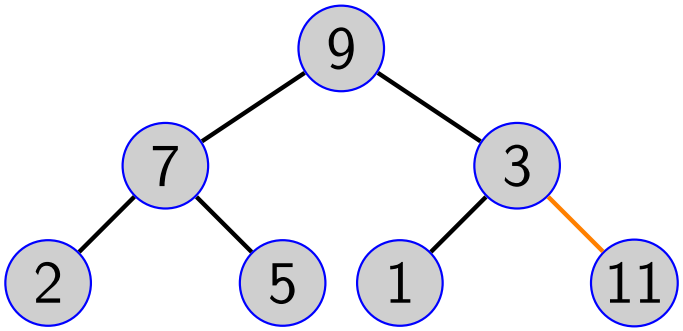
Die Funktion Insert

- Wie navigiert man in einem Heap H?
  Wenn Knoten v in Position i gespeichert ist,

  dann ist das linke Kind v_L in Position $2 \cdot i$, das rechte Kind in Position $2 \cdot i + 1$ und der Vater von v in Position $\lfloor i/2 \rfloor$ gespeichert.
- Wenn wir die Priorität p einfügen wollen, liegt es nahe, p auf der ersten freien Position abzulegen. Wir erhöhen also den Zähler n für die Anzahl der Elemente im Heap um eins: $n = n + 1$, und speichern danach die neue Priorität ab: $H[n] = p$
  - Der neue Baum hat Heap-Struktur,
  - aber die Heap-Ordnung ist möglicherweise verletzt.

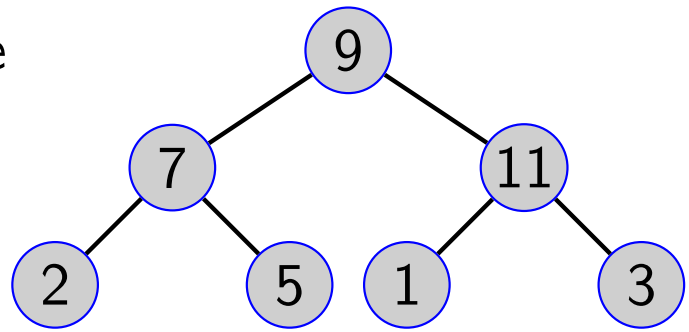
Wie kann die Heap-Ordnung kostengünstig repariert werden?

Wir fügen die Priorität 11 ein

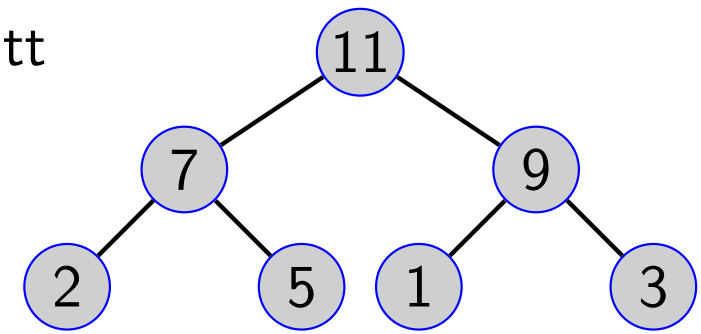


Nach dem Anhängen von 11
ist die Heap-Ordnung verletzt.

Darum rutscht die
11 nach oben:



Und ein weiterer Vertauschungsschritt
repariert die Heap-Ordnung:



Die Repair_up Prozedur

Die Klasse heap enthalte die Funktion repair_up.

```
void heap::repair_up (int wo){
   int p = H[wo];
   while ((wo > 1) && (H[wo/2] < p)){
      H[wo] = H[wo/2];
      wo = wo/2;
   }
   H[wo] = p;
}
```

- Wir verschieben die Priorität solange nach oben, bis
  - entweder die Priorität des Vaters mindestens so groß ist
  - oder bis wir die Wurzel erreicht haben.
- Wie groß ist der Aufwand? Höchstens proportional zur Tiefe des Baums!