Definition: Ungerichteter Graph

Definition: Ungerichteter Graph

Ein ungerichteter Graph ist ein Paar $G = (V, E)$ mit

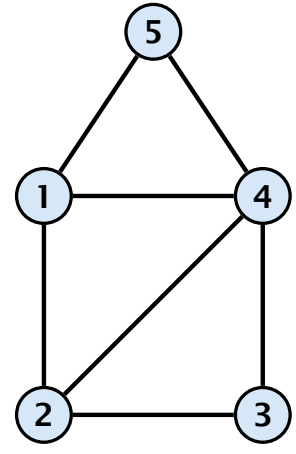
- V endliche Menge der Knoten
- $E \subseteq \{\{u, v\} : u, v \in V\}$ Menge der Kanten

auf Englisch:

- Knoten = vertices
- Kanten = edges

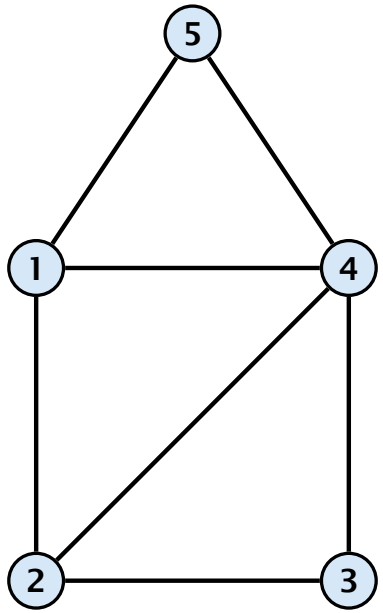
es ist $\{u, v\} = \{v, u\}$, d.h. Richtung der Kante spielt keine Rolle

Beispiel:



Manchmal erlaubt man auch Schleifen (self-loops); dann muss man E als Multimenge modellieren.

Ungerichteter Graph: Beispiel



- Graph $G_u = (V_u, E_u)$
- Knoten $V_u = \{1, 2, 3, 4, 5\}$
- Kanten $E_u = \{\{1, 2\}, \{1, 4\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$

Definition: Gerichteter Graph

Definition: Gerichteter Graph

Ein gerichteter Graph ist ein Paar $G = (V, E)$ mit

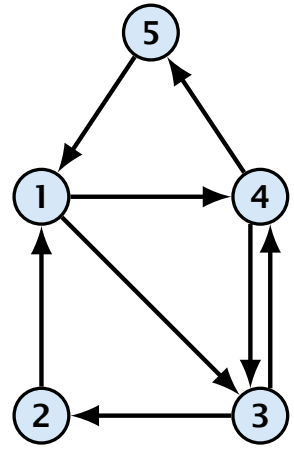
- V endliche Menge der Knoten
- $E \subseteq V \times V$ Menge der Kanten

$E \subseteq \{(u, v) : u, v \in V\} = V \times V$

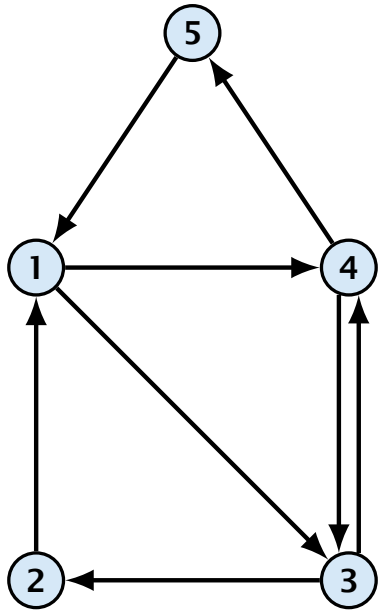
es ist $(u, v) \neq (v, u)$, d.h. Richtung der Kante spielt eine Rolle

hier sind Schleifen möglich, d.h. Kanten der Form (u, u) für $u \in V$

Beispiel:



Gerichteter Graph: Beispiel



- Graph $G_g = (V_g, E_g)$
- Knoten $V_g = \{1, 2, 3, 4, 5\}$
- Kanten
  $E_g = \{(1,3), (1,4), (2,1), (3,2), (3,4), (4,3), (4,5), (5,1)\}$

Definition: Gewichteter Graph

Definition: Gewichteter Graph

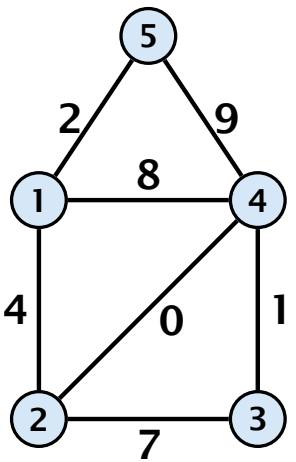
Ein gewichteter Graph ist ein Graph $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to \mathbb{R}$.

der Graph G kann gerichtet oder ungerichtet sein

je nach Anwendung kann ein verschiedener Wertebereich für die Funktion w gewählt werden

- z.B. $\mathbb{R}$ oder $\mathbb{N}_0$

Beispiel:



Eigenschaften von Graphen I

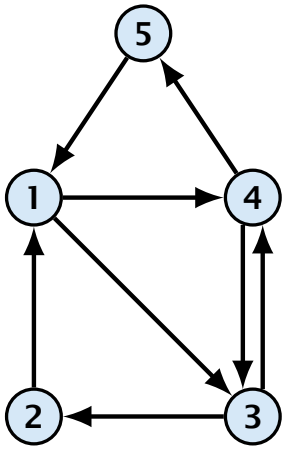
Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

- Ist $(u, v) \in E$ bzw. $\{u, v\} \in E$ für $u, v \in V$, so heißt v adjazent zu u.
- Ein Knoten v und eine Kante $e = (x, v)$ or $e = \{x, v\}$ heißen inzident.
- Sei G gerichteter Graph:
  - die Anzahl der eintretenden Kanten in v heißt Eingangsgrad von v, $\text{indeg}(v) = \left|\{v' : (v', v) \in E\}\right|$
  - die Anzahl der austretetenden Kanten heißt Ausgangsgrad von v, $\text{outdeg}(v) = \left|\{v' : (v, v') \in E\}\right|$
- Sei G ungerichteter Graph:
  - die Anzahl der eintretenden bzw. austretenden Kanten von v heißt Grad (englisch: degree) von v oder kurz $\deg(v)$.

Eigenschaften von Graphen: Beispiel

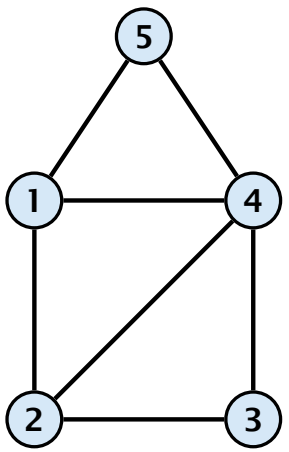
Beispiel gerichteter Graph:

- Knoten 1 ist adjazent zu Knoten 2
- Knoten 2 ist adjazent zu Knoten 3
- $\text{outdeg}(4) = 2$, $\text{indeg}(4) = 2$
- $\text{indeg}(2) = 1$, $\text{outdeg}(2) = 1$



Beispiel ungerichteter Graph:

- Knoten 4 ist adjazent zu Knoten 2
- Knoten 2 ist adjazent zu Knoten 4
- $\deg(2) = 3$
- $\deg(5) = 2$



Eigenschaften von Graphen II

Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

- Seien $v, v' \in V$. Ein Pfad von v nach v' ist eine Folge von Knoten $(v_0, v_1, \ldots, v_k) \subset V$ mit
  - $v_0 = v$, $v_k = v'$
  - $(v_i, v_{i+1}) \in E$ bzw. $\{v_i, v_{i+1}\} \in E$ für $i = 0, \ldots, k-1$

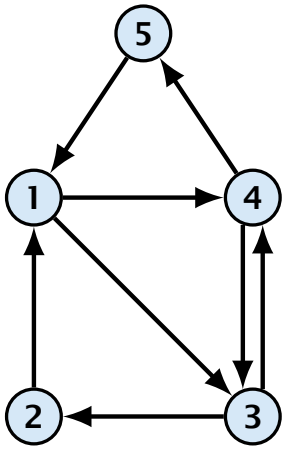
  k heißt Länge des Pfades.
- Ein Pfad heißt einfach, falls alle Knoten des Pfades paarweise verschieden sind.

Gibt es einen Pfad von u nach v, so heißt v erreichbar von u.

Eigenschaften von Graphen II: Beispiel

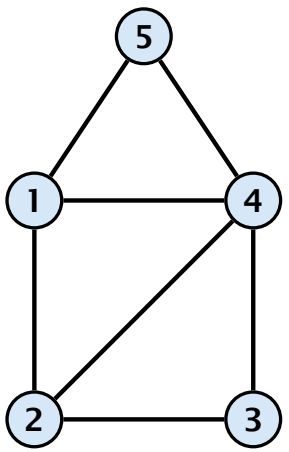
Beispiel gerichteter Graph:

- $(2, 1, 3)$ ist ein einfacher Pfad der Länge 2
- $(1, 4, 5, 1)$ ist ein Pfad der Länge 3, aber nicht einfach
- 5 ist erreichbar von 1



Beispiel ungerichteter Graph:

- $(5, 2, 4, 3, 2)$ ist ein Pfad der Länge 4, aber nicht einfach
- $(1, 2, 3, 4)$ ist ein einfacher Pfad der Länge 3
- 3 ist erreichbar von 1



Eigenschaften von Graphen III

Sei $G = (V, E)$ Graph.

- Ein Pfad $(v_0, \ldots, v_k)$ heißt Zyklus, falls $v_0 = v_k$.
- Ein Zyklus $(v_0, \ldots, v_k)$ heißt Kreis, falls $v_1, \ldots, v_k$ paarweise verschieden sind.

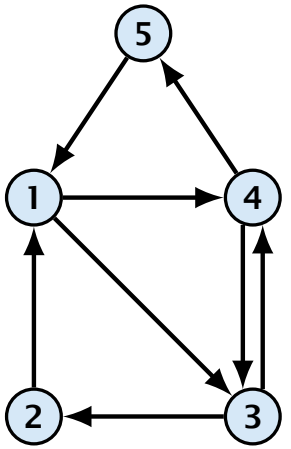
Zyklen der Länge 1 oder 2 heißen trivial und werden häufig nicht betrachtet.

Ein Graph ohne Zyklen heißt azyklisch.

Eigenschaften von Graphen III: Beispiel

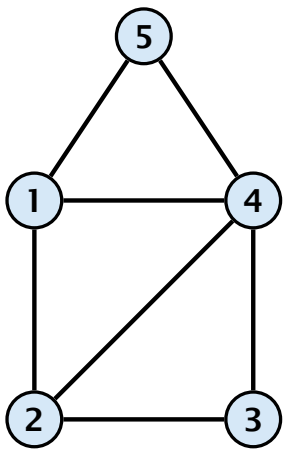
Beispiel gerichteter Graph:

- $(2, 1, 3, 4, 3, 2)$ ist ein Zyklus, aber nicht einfach
- $(2, 1, 3, 2)$ ist ein einfacher Zyklus



Beispiel ungerichteter Graph:

- $(2, 1, 3, 2)$ ist ein Zyklus
- $(1, 5, 4, 1)$ ist ein Zyklus



Eigenschaften von Graphen IV

Sei $G = (V, E)$ gerichteter Graph.

- G heißt stark zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine starke Zusammenhangskomponente von G ist ein maximaler zusammenhängender Untergraph von G.
  - alternativ: Äquivalenzklassen der Knoten bezüglich Relation "gegenseitig erreichbar"

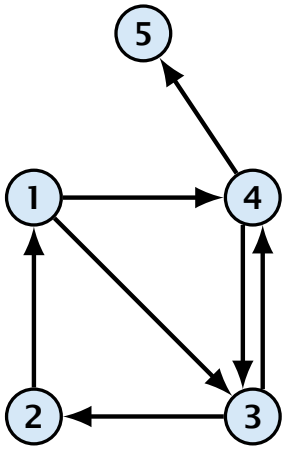
Sei $G = (V, E)$ ungerichteter Graph.

- G heißt zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine Zusammenhangskomponente von G ist ein maximaler zusammenhängender Untergraph von G.
  - alternativ: Äquivalenzklassen der Knoten bezüglich Relation "erreichbar von"

Eigenschaften von Graphen IV: Beispiel

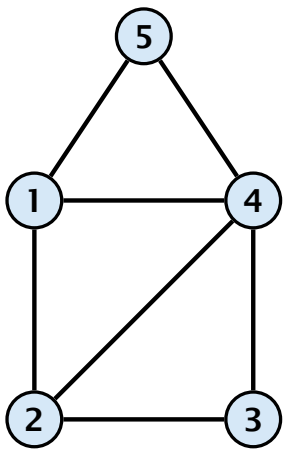
Beispiel gerichteter Graph:

- Graph ist nicht stark zusammenhängend (z.B. 3 nicht erreichbar von 5)
- starke Zusammenhangskomponenten: $\{1,2,3,4\}$ und $\{5\}$



Beispiel ungerichteter Graph:

- Graph ist zusammenhängend
- nur eine Zusammenhangskomponente: $\{1,2,3,4,5\}$



Darstellung von Graphen: Adjazenzmatrizen

Adjazenzmatrix

Sei $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$. Die Adjazenzmatrix von G speichert die vorhandenen Kanten in einer $n \times n$ Matrix $A \in \mathbb{R}^{n \times n}$ mit

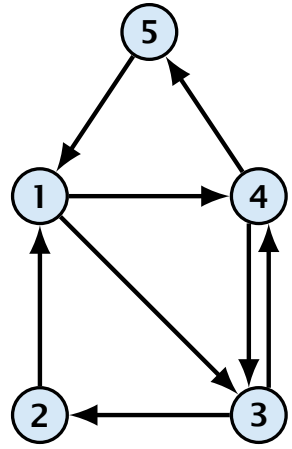
- $A(i, j) = 1$ falls Kante von Knoten v_i zu v_j existiert
- $A(i, j) = 0$ falls keine Kante von Knoten v_i zu v_j existiert

für $i, j \in \{1, \ldots, n\}$.

Eigenschaften von Graphen: Adjazenzmatrizen

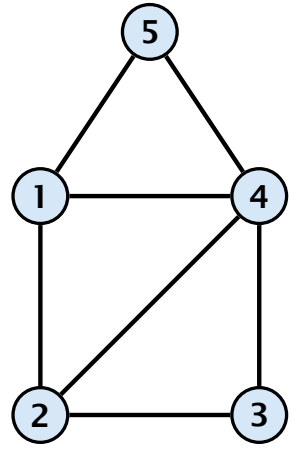
Beispiel gerichteter Graph:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$



Beispiel ungerichteter Graph:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$



Adjazenzmatrizen: Eigenschaften

Eigenschaften von Adjazenzmatrizen zu Graph $G = (V, E)$

- sinnvoll wenn der Graph nahezu vollständig ist (d.h. fast alle möglichen Kanten tatsächlich in E liegen)
- Speicherkomplexität: $O(|V|^2)$
- bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch
- bei gewichteten Graphen kann man statt der 1 in der Matrix das Gewicht der Kante eintragen

Darstellung von Graphen: Adjazenzlisten

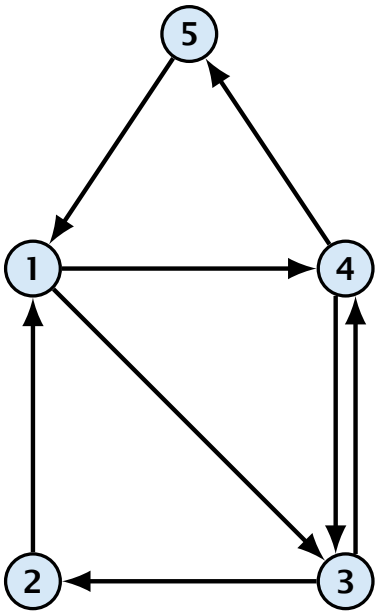
Adjazenzliste

Sei $G = (V, E)$ gerichteter Graph. Eine Adjazenzliste von G sind $|V| + 1$ verkettete Listen, so daß

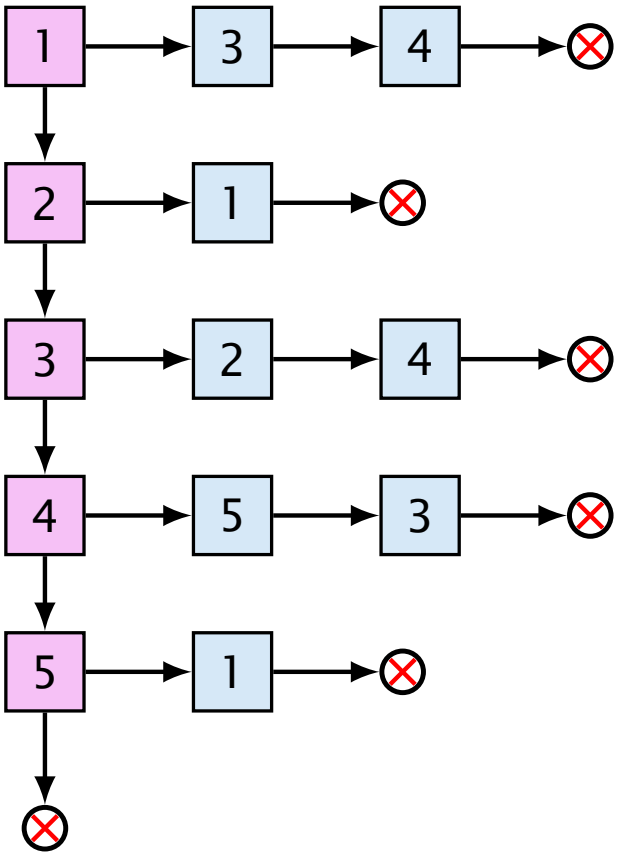
- die erste Liste alle Knoten enthält
- für jeden Knoten v eine Liste angelegt wird mit allen Knoten, die durch eine von v austretende Kante zu erreichen sind

Adjazenzliste: Beispiel

Graph



Adjazenzliste



Adjazenzliste: Eigenschaften

Eigenschaften von Adjazenzlisten zu Graph $G = (V, E)$

- sinnvoll bei dünn besetzten Graphen mit wenigen Kanten
- Speicherkomplexität: $O(|V| + |E|)$
- bei **ungerichteten** Graphen gleiches Verfahren
  - allerdings muß jede Kante zweimal gespeichert werden
- bei **gewichteten** Graphen kann man die Gewichte mit in den verketteten Listen der jeweiligen Knoten speichern

Komplexität der Darstellungen

Sei $G = (V, E)$ Graph.

Operation	Adjazenzmatrix	Adjazenzliste
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$
Kante löschen	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|)$
Knoten einfügen	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(1)$
Knoten löschen	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| + \deg(v))$

- falls Größe im Vorhinein bekannt, kann Knoten löschen/einfügen bei Adjanzenzmatrix effizienter implementiert werden
- Löschen von Knoten ist immer aufwendig, da auch alle Kanten von/zu diesem Knoten gelöscht werden müssen

Algorithmen auf Graphen

Ausblick auf Algorithmen auf Graphen:

- Traversierung (Durchlaufen) von allen Knoten
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- kürzester Pfad zwischen Knoten in Graphen
- minimaler Spannbaum (minimum spanning tree, MST)

Bäume

Bäume sind alltägliches Mittel zur Strukturierung:

- Stammbaum
- Hierarchie in Unternehmen
- Systematik in der Biologie
- etc.

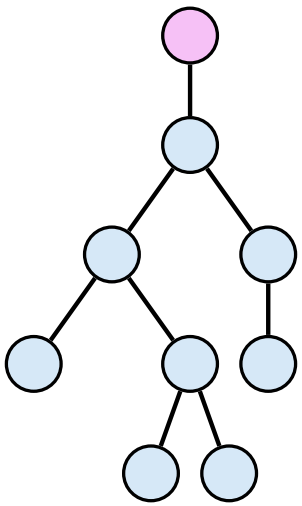
In Informatik:

- Bäume sind spezielle Graphen
- Wurzel oben!

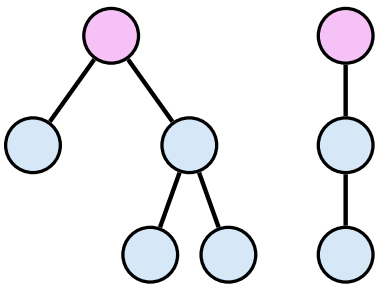
Definition Wald/Baum

Definition: Wald und Baum

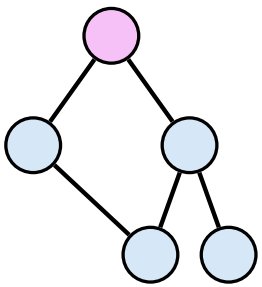
- Ein azyklischer ungerichteter Graph heißt auch Wald.
- Ein zusammenhängender, azyklischer ungerichteter Graph heißt auch Baum.



Baum



Wald



kein Baum

Eigenschaften von Bäumen

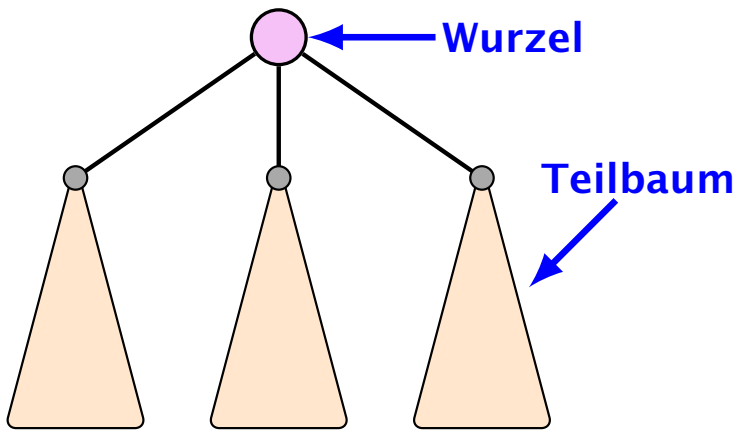
Sei $G = (V, E)$ ein Baum.

- jedes Paar von Knoten $u, v \in V$ ist durch einen einzigen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante aus E **entfernt** wird, ist G nicht mehr zusammenhängend
- G ist azyklisch, aber wenn eine Kante zu E **hinzugefügt** wird, ist G nicht mehr azyklisch
- es gilt $|E| = |V| - 1$

Wurzel von Bäumen

Sei $G = (V, E)$ ein Baum.

- genau ein Knoten $w \in V$ wird als Wurzel ausgezeichnet
- entfernt man w erhält man einen Wald von Teilbäumen

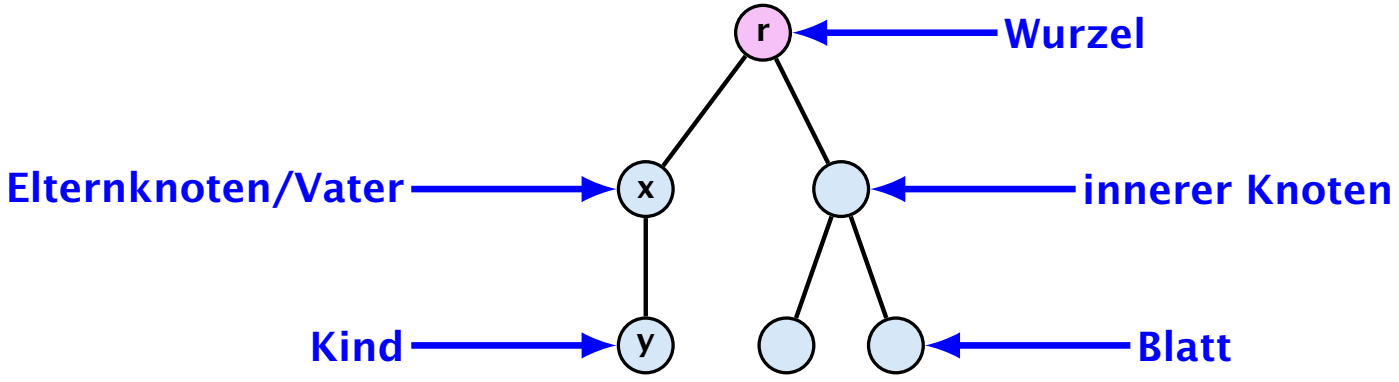


Hinweis: manchmal wird zwischen “freiem” und “gewurzeltem” Baum unterschieden!

Weitere Begriffe bei Bäumen I

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

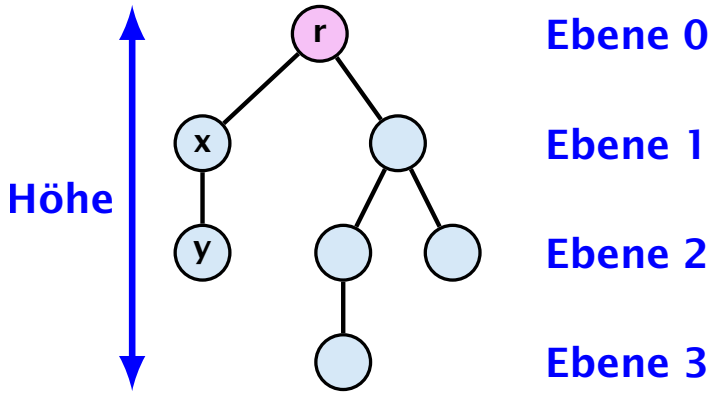
- jeder Knoten $v \in V$ mit $v \neq w$ ist mit genau einer Kante mit seinem Elternknoten $x \in V$ (oder: direkter Vorgänger) verbunden
- v wird dann als Kind (oder: direkter Nachfolger) von $x \in V$ bezeichnet
- ein Knoten ohne Kinder heißt Blatt, alle anderen Knoten heißen innere Knoten



Weitere Begriffe bei Bäumen II

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

- Anzahl der Kinder von Knoten $x \in V$ heißt auch Grad von x. (Achtung: Grad in Graph G ist anders definiert!)
- Länge des Pfades von Wurzel w zu Knoten $x \in V$ heißt Tiefe von x
- alle Knoten gleicher Tiefe bilden eine Ebene des Baumes G
- maximale Tiefe eines Knotens heißt Höhe des Baumes. (manchmal ist Höhe auch als Anzahl der Ebenen definiert)

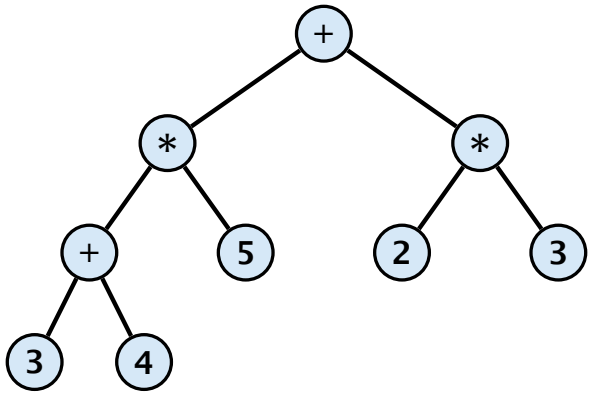


Bäume: Beispiele

arithmetischer Ausdruck

$$(3 + 4) * 5 + 2 * 3$$

repräsentiert als Baum:



hierarchisches Dateisystem

- Windows z.B. “C:\”
- Unix “/”

Suchbaum → später in der Vorlesung

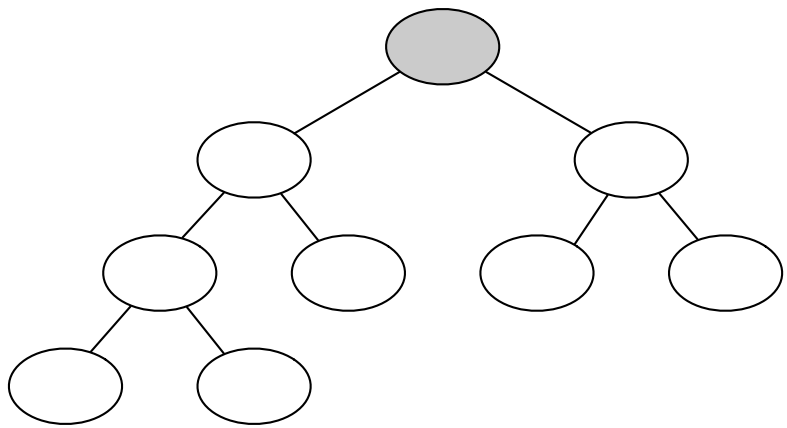
Besondere Bäume

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

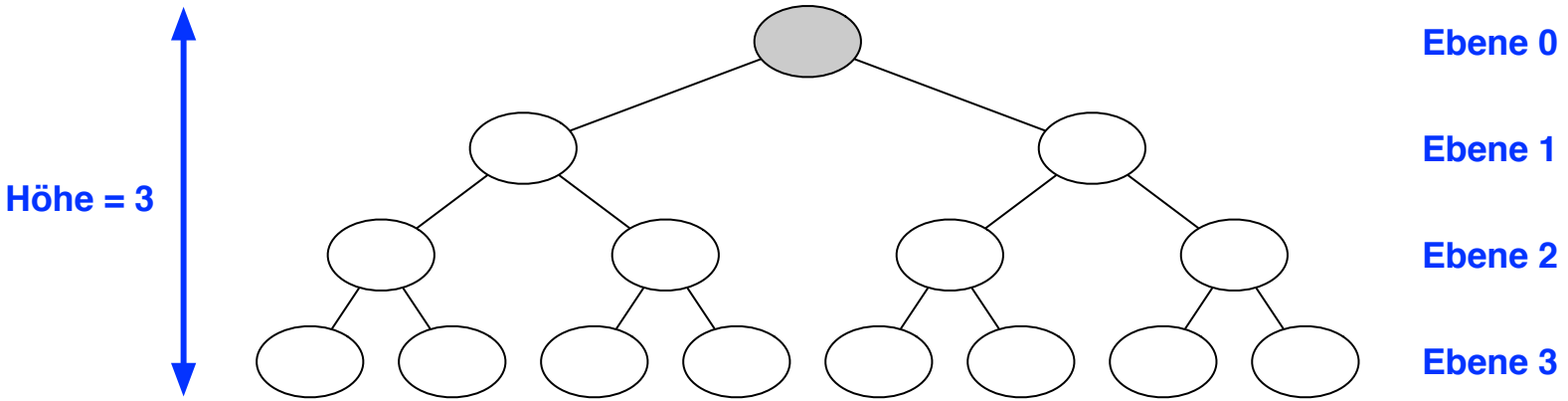
- sind die Kinder jedes Knotens in bestimmter Reihenfolge angeordnet, heißt G geordneter Baum
- ist die Anzahl n der Kinder jedes Knotens vorgegeben, heißt G n-ärer Baum

Wichtiger Spezialfall:

- ist G geordnet und hat jeder Knoten maximal zwei Kinder, heißt G Binärbaum



Beispiel: Binärbaum



Binärbaum mit Höhe 3, 8 Blättern und 7 inneren Knoten.

- Binärbaum heißt vollständig, wenn jede Ebene die maximale Anzahl an Knoten enthält
- ein vollständiger Binärbaum der Höhe k hat $2^{k+1} - 1$ Knoten, davon 2^k Blätter
- Beweis per Induktion

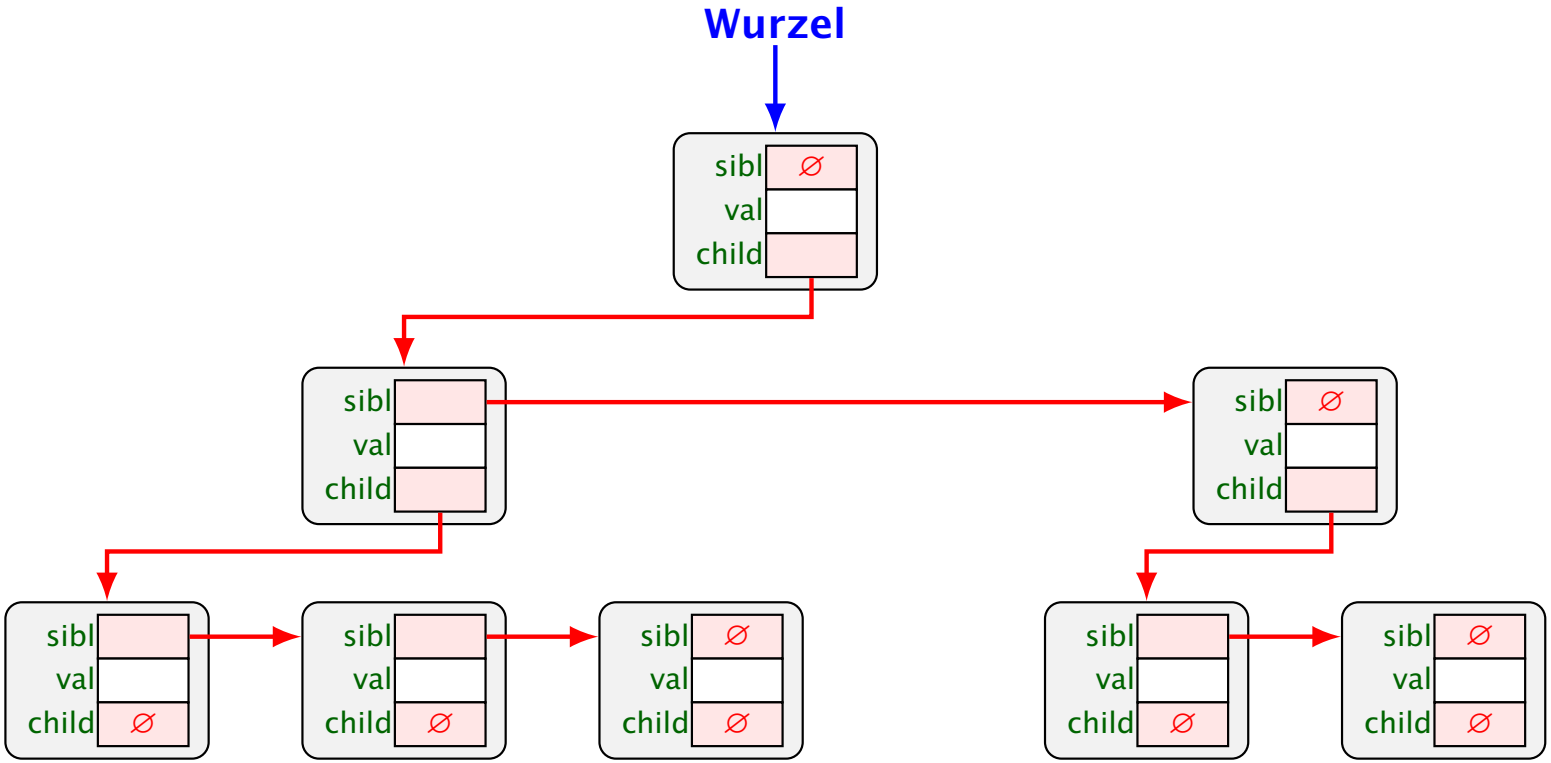
Darstellung von Bäumen: als Graph

Bäume sind Graphen → Darstellung als

- Adjazenzmatrix
- Adjazenzliste

⇒ leider meist nicht effizient (sowohl Laufzeit als auch Speicher)

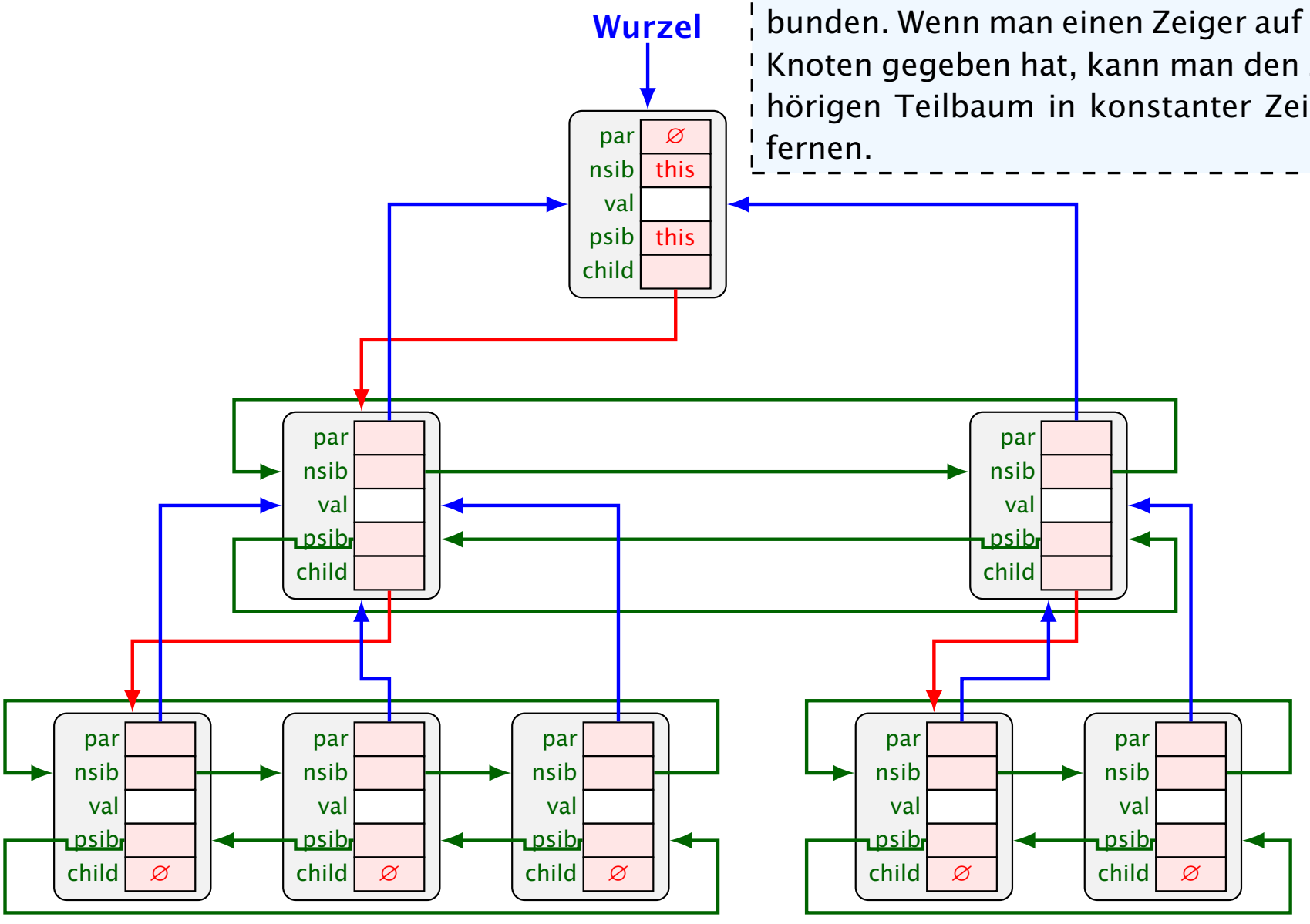
Darstellung von Bäumen: verkettete Liste



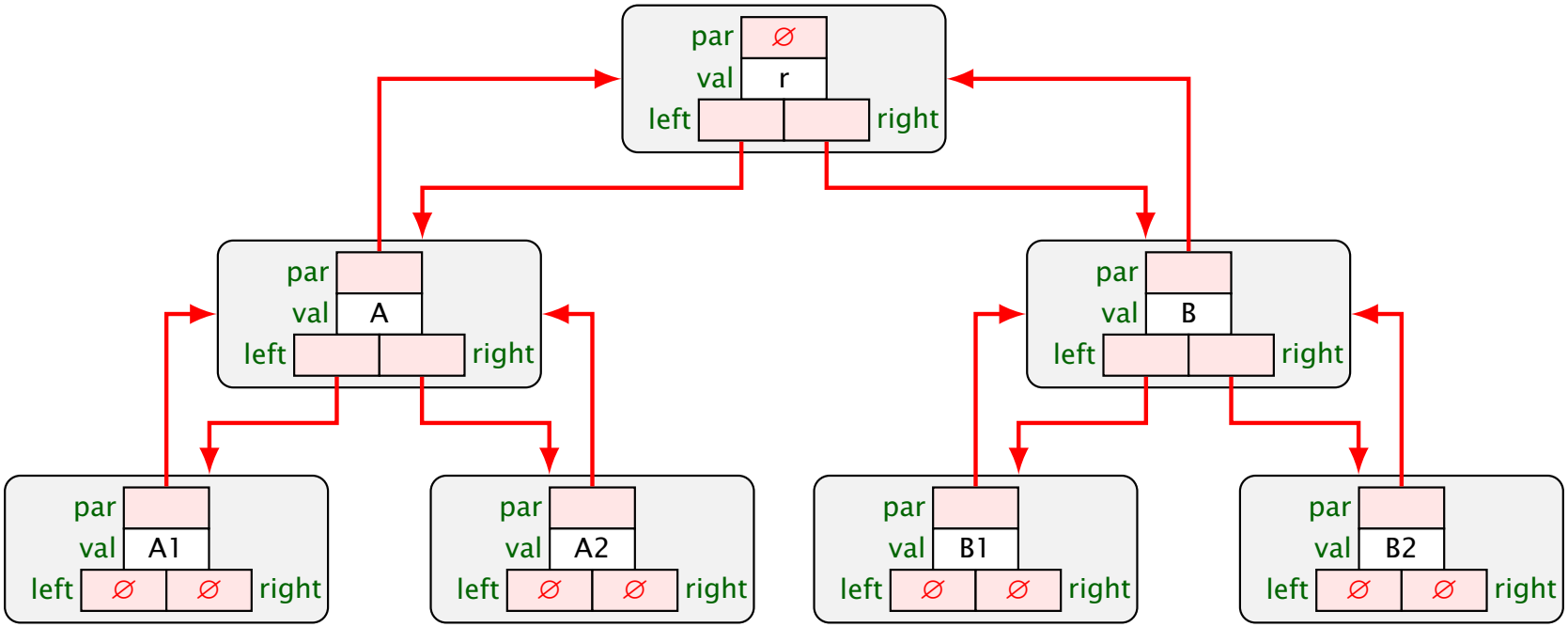
Nachteil: nur Navigation nach unten möglich.

Darstellung von Bäumen: doppelt verkettet

Geschwister sind über zyklische Liste verbunden. Wenn man einen Zeiger auf einen Knoten gegeben hat, kann man den zugehörigen Teilbaum in konstanter Zeit entfernen.



Darstellung von Binärbäumen



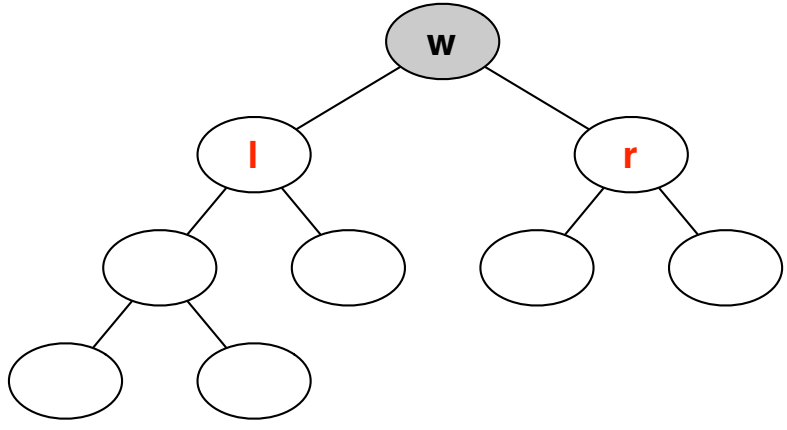
Bei Binärbäumen hat man üblicherweise für beide Nachfolger eine explizite Referenz.

Traversierung von Binärbäumen

Sei $G = (V, E)$ Binärbaum.

In **welcher Reihenfolge** durchläuft man G?

- **Wurzel** zuerst
- danach linker oder rechter Kind-Knoten l bzw. r?
- falls l: danach Kindknoten von l oder zuerst r?
- falls r: danach Kindknoten von r oder zuerst l?



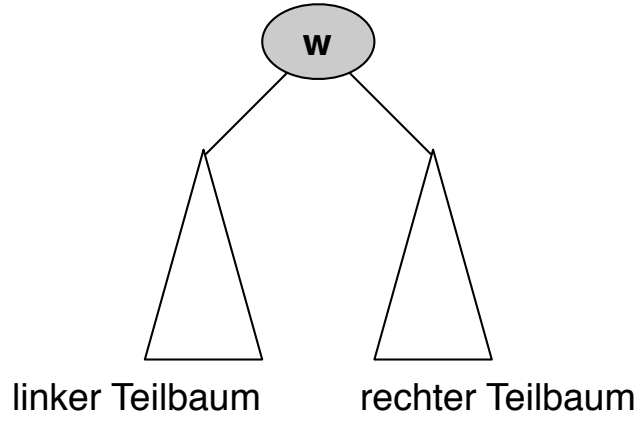
⇒ falls zuerst in die Tiefe: Depth-first search (DFS)

⇒ falls zuerst in die Breite: Breadth-first search (BFS)

DFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

Tiefensuche (Depth-first search, DFS) gibt es in 3 Varianten:



1. Pre-order Reihenfolge
   - besuche Wurzel
   - durchlaufe linken Teilbaum
   - durchlaufe rechten Teilbaum
2. In-order Reihenfolge
   - durchlaufe linken Teilbaum
   - besuche Wurzel
   - durchlaufe rechten Teilbaum
3. Post-order Reihenfolge
   - durchlaufe linken Teilbaum
   - durchlaufe rechten Teilbaum
   - besuche Wurzel

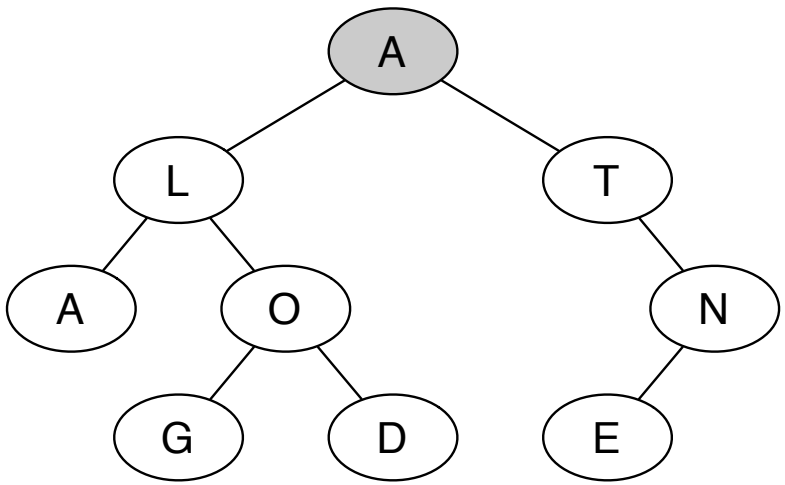
Pre-order Traversierung

Pre-order Reihenfolge:

- besuche Wurzel
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

Beispiel:

A, L, A, O, G, D, T, N, E



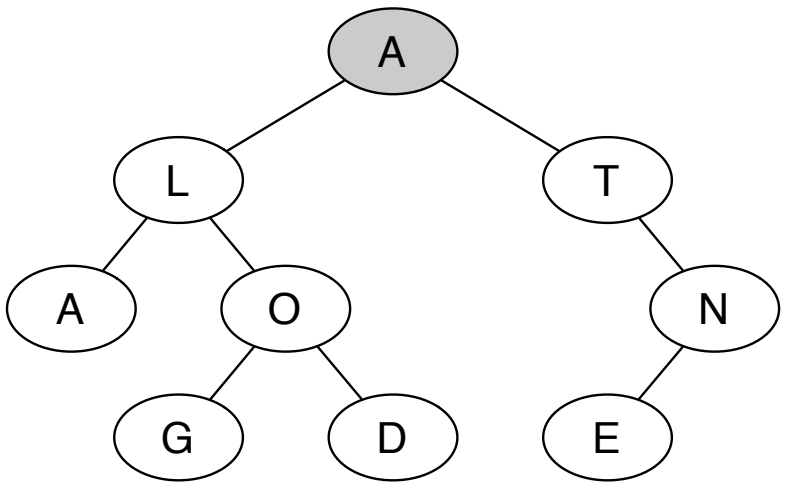
In-order Traversierung

Inorder Reihenfolge:

- durchlaufe linken Teilbaum
- besuche Wurzel
- durchlaufe rechten Teilbaum

Beispiel:

A, L, G, O, D, A, T, E, N



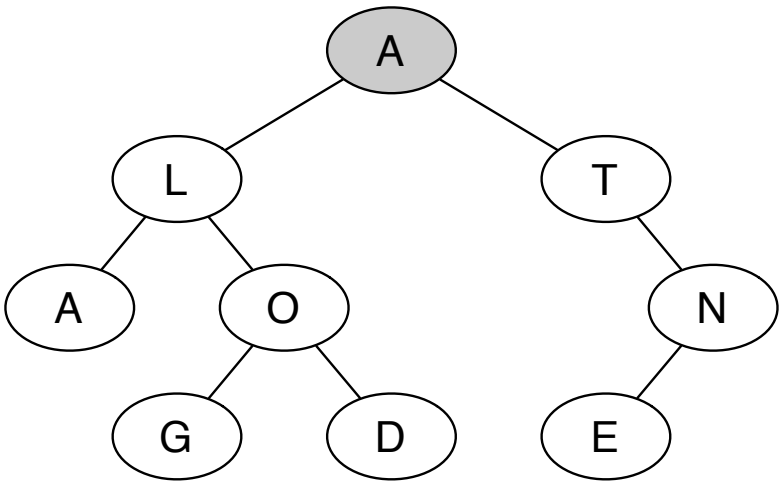
Post-order Traversierung

Postorder Reihenfolge:

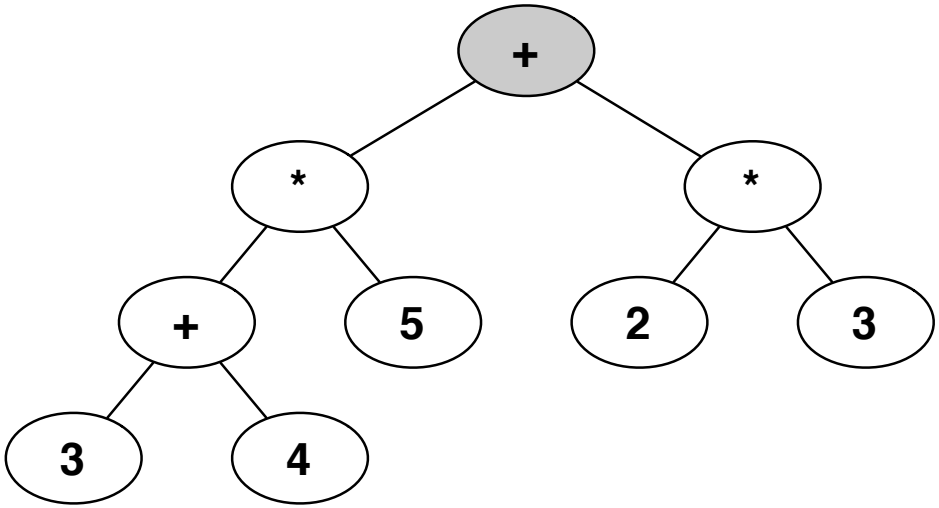
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

Beispiel:

A, G, D, O, L, E, N, T, A



Beispiel: Arithmetischer Term

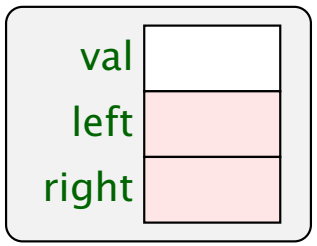


Traversierung:

- **Pre-order:** $+ * + 3\ 4\ 5 * 2\ 3$
- **In-order:** $3 + 4 * 5 + 2 * 3$
- **Post-order:** $3\ 4 + 5 * 2\ 3 * +$

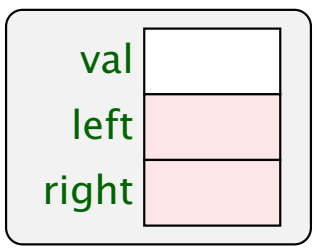
Implementierung DFS

```
1 preorder(TreeNode* v)
2     if (v == NULL)
3         return
4     print(v->val);
5     preorder(v->left);
6     preorder(v->right);
```



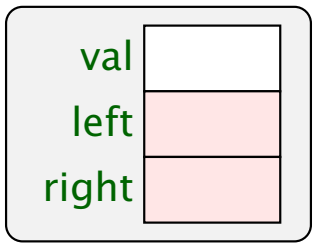
Implementierung DFS

```
1 inorder(TreeNode* v)
2     if (v == NULL)
3         return
4     preorder(v->left);
5     print(v->val);
6     preorder(v->right);
```



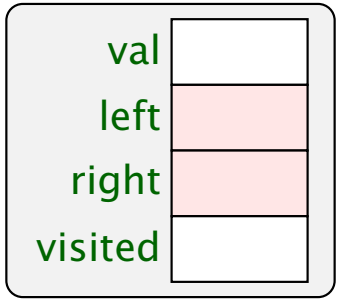
Implementierung DFS

```
1 postorder(TreeNode* v)
2     if (v == NULL)
3         return
4     postorder(v->left);
5     postorder(v->right);
6     print(v->val);
```



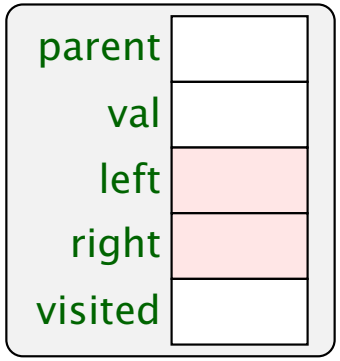
Implementierung DFS mit Stack

```
1  postorder(TreeNode* v)
2      stack.push(v);
3      while (!stack.empty())
4          v = stack.top();
5          if (v->left) && !v->left->visited)
6              stack.push(v->left);
7          else if (v->right && !v->right->visited)
8              stack.push(v->right);
9          else
10             print(v->val);
11             v->visited = true;
12             stack.pop();
```



Implementierung DFS ohne Stack

```
1  postorder(TreeNode* v)
2      TreeNode* k = v;
3      while (true)
4          if (k->left && !k->left->visited)
5              k = k->left;
6          else if (k->right && !k->right->visited)
7              k = k->right;
8          else
9              print(k->val);
10             k->visited = true;
11             if (k == v) break;
12             else k = k->parent;
```

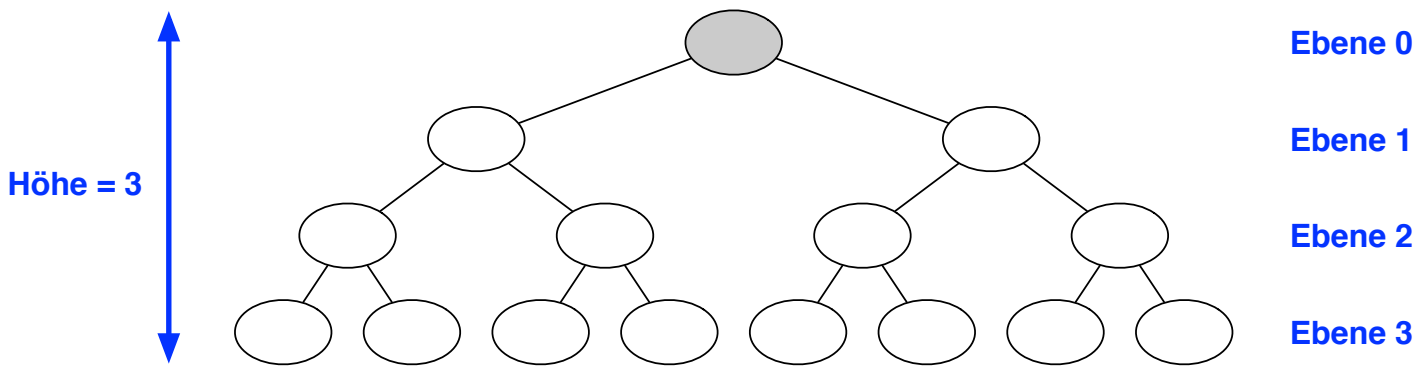


BFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

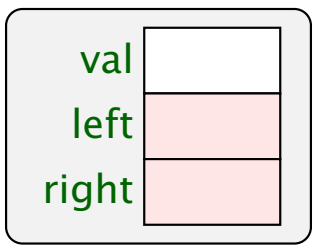
Breitensuche (Breadth-first search, BFS):

- besuche Wurzel
- für alle Ebenen von 1 bis Höhe
  - besuche alle Knoten aktueller Ebene



Implementierung BFS Traversierung

```
1 bfs(TreeNode* v)
2     queue.enqueue(v);
3     while (!queue.empty())
4         v = queue.dequeue();
5         print(v.val);
6         if (v->left)
7             queue.enqueue(v->left);
8         if (v->right)
9             queue.dequeue(v->right);
```

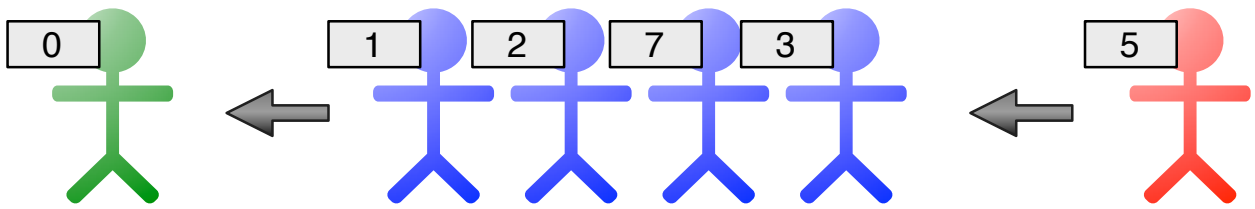


Definition Priority Queue

Definition Priority Queue

Eine Priority Queue ist ein abstrakter Datentyp. Sie beschreibt einen **Queue-artigen** Datentyp für eine Menge von Elementen mit **zugeordnetem Schlüssel** und unterstützt die Operationen

- Einfügen von Element mit Schlüssel in die Queue,
- Entfernen von Element mit minimalem Schlüssel aus der Queue,
- Ansehen des Elementes mit minimalem Schlüssel in der Queue.



- entsprechend gibt es auch eine Priority Queue mit Entfernen/Ansehen von Elememt mit **maximalem Schlüssel**

Definition Priority Queue (abstrakter)

Priority Queue P ist ein abstrakter Datentyp mit Operationen

- **insert(P, x)** wobei x ein Element
- **extractMin(P)** liefert ein Element
- **minimum(P)** liefert ein Element
- **isEmpty(P)** liefert `true` or `false`
- **initialize** liefert eine Priority Queue Instanz

und mit Bedingungen

- **isEmpty(initialize()) == `true`**
- **isEmpty(insert(P, x)) == `false`**
- **minimum(initialize())** ist nicht erlaubt (Fehler)
- **extractMin(initialize())** ist nicht erlaubt (Fehler)

(Fortsetzung nächste Folie)

Definition Priority Queue (abstrakter)

Fortsetzung Bedingungen Priority Queue P:

- **minimum(insert(P, x))** liefert zurück
  - falls `P == initialize()`, dann `x`
  - sonst: `min(x, minimum(P))`

- **extractMin(insert(P, x))**
  - falls `x == minimum(insert(P, x))`, dann liefert es `x` zurück und hinterlässt `P` im Originalzustand
  - sonst liefert es `extractMin(P)` zurück und hinterlässt `P` im Zustand `insert(extractMin(P), x)`

(entsprechend für die Priority Queue mit maximalem Schlüssel)

Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k)**:
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k):**
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

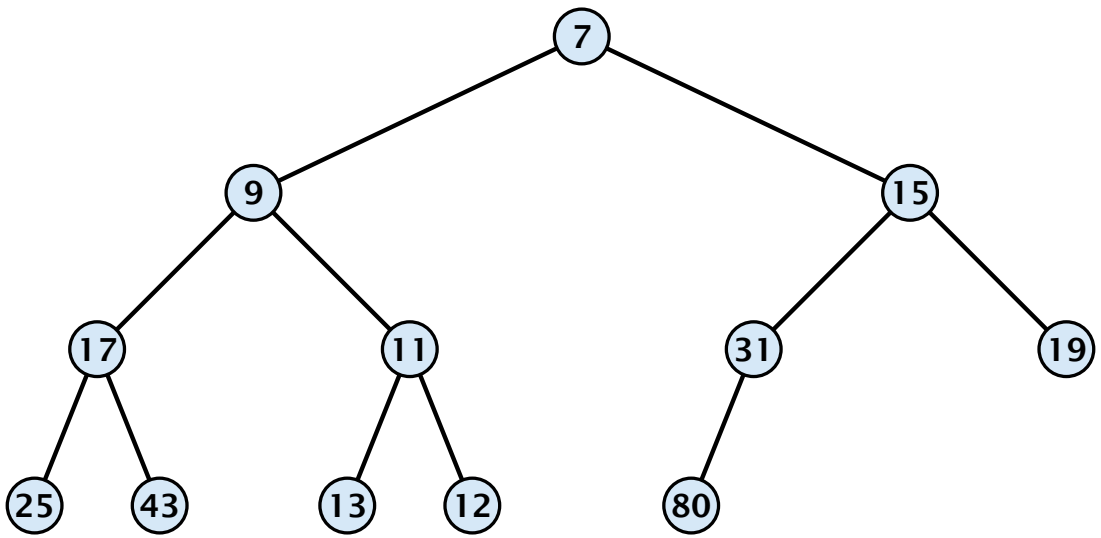
- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k):**
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

Definition Priority Queue

Eine addressierbare Priority Queue unterstützt zusätzlich:

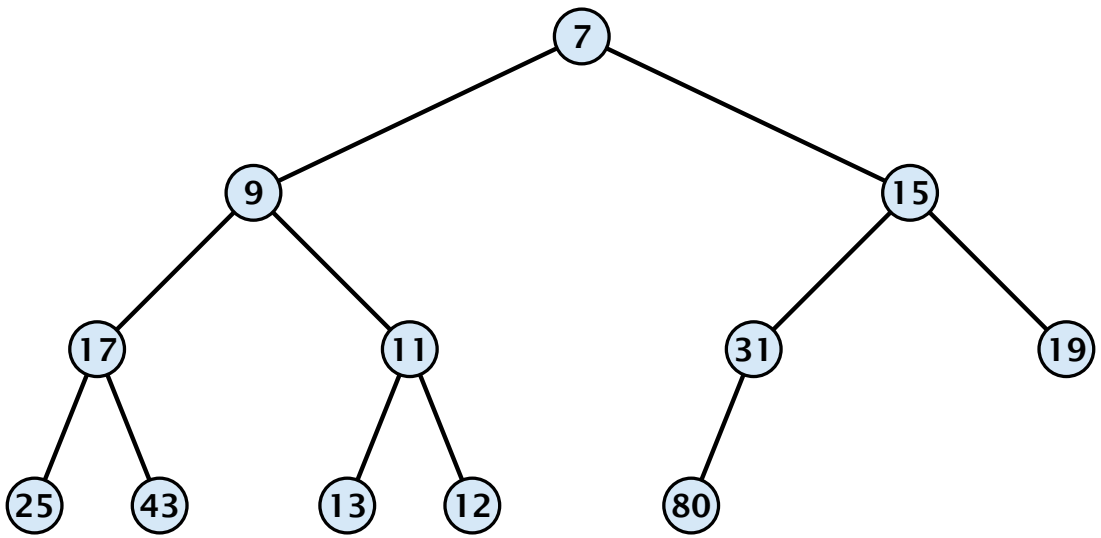
- **handle insert(P,x):**
  Fügt x; gibt ein handle zuück mit dem man später noch auf das Objekt zugreifen kann.
- **delete(P, h):**
  Entfernt, das durch handle h referenzierte Objekt.
- **decrease-key(P, h, k)**:
  Ändert den Schlüssel des Objektes, das durch h referenziert wird auf k. Erfordert k<h.

7.3 Priority Queues



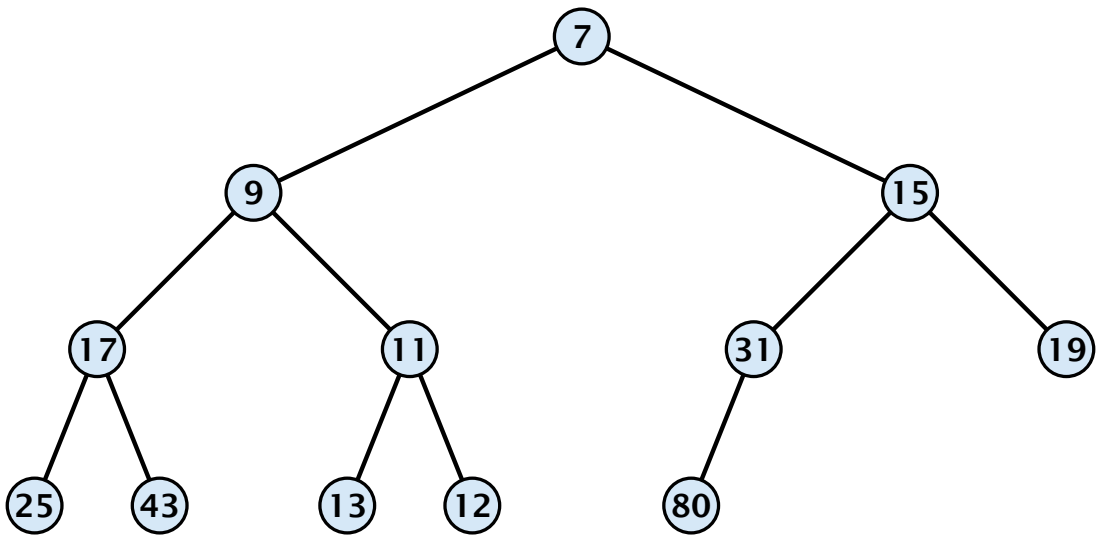
7.3 Priority Queues

- **Idee:** Speichere Elemente in einem fast vollständigen Binärbaum.



7.3 Priority Queues

- **Idee:** Speichere Elemente in einem fast vollständigen Binärbaum.
- Heapeigenschaft: Der Schlüssel eines Elements ist nicht größer als der Schlüssel eines Kindes.



Binäre Heaps

Operationen:

- **minimum():** gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** überprüfe ob Zeiger auf Wurzel NULL ist. Zeit $\mathcal{O}(1)$.

Binäre Heaps

Operationen:

- **minimum():** gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** überprüfe ob Zeiger auf Wurzel NULL ist. Zeit $\mathcal{O}(1)$.

Binäre Heaps

Operationen:

- **minimum():** gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** überprüfe ob Zeiger auf Wurzel NULL ist. Zeit $\mathcal{O}(1)$.

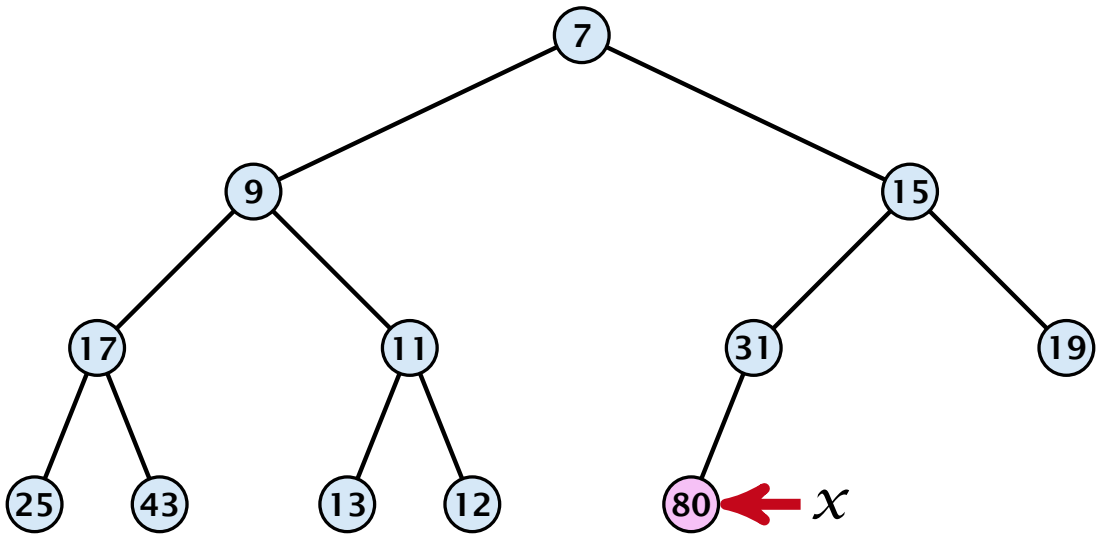
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Berechne Vorgänger von x (letztes Element wenn x gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

  gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

  wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch rechte Kanten.



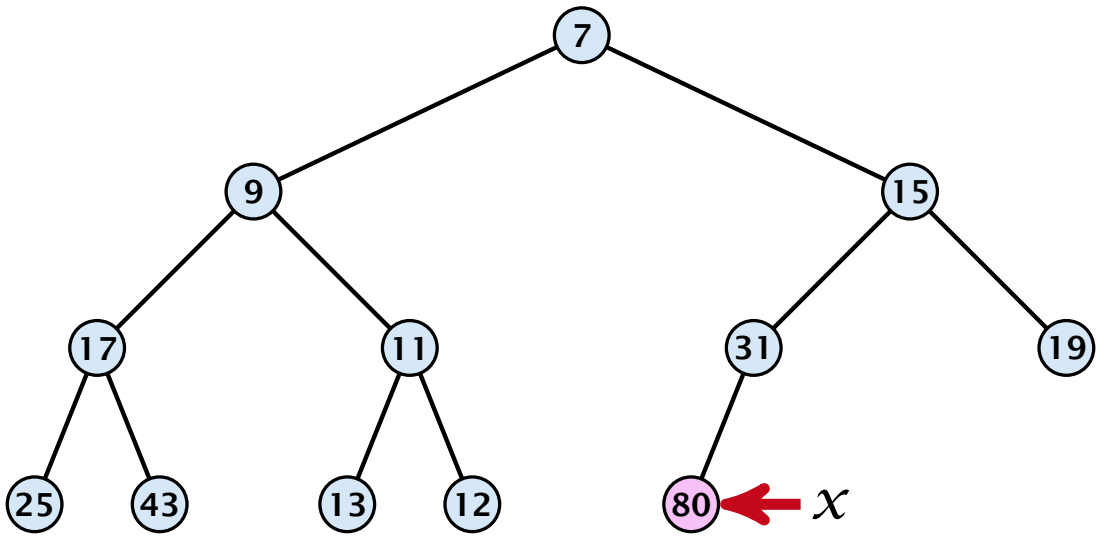
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Berechne Vorgänger von x (letztes Element wenn x gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur nocht rechte Kanten.



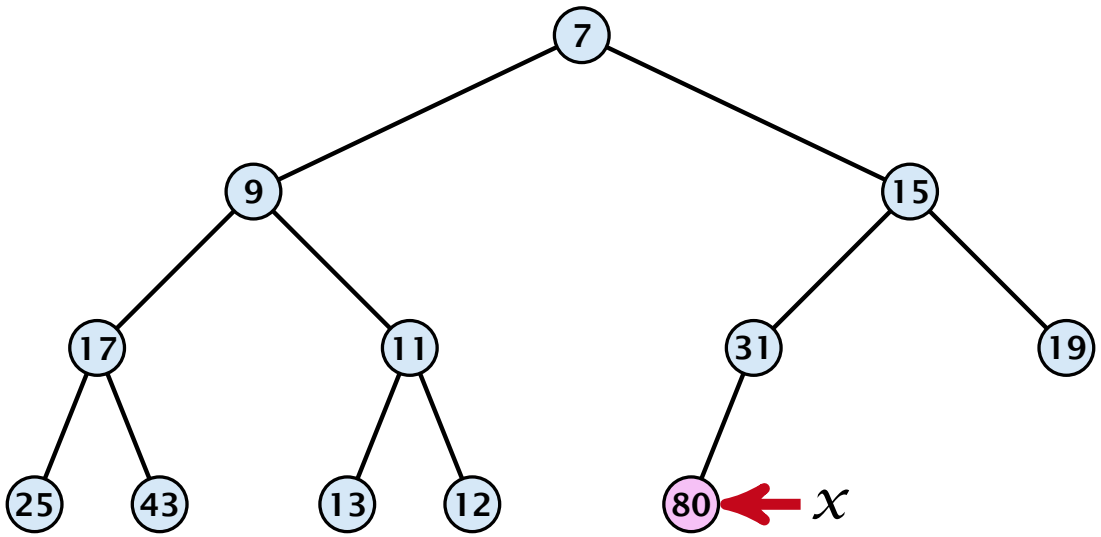
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Berechne Vorgänger von x (letztes Element wenn x gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

  gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

  wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur nocht rechte Kanten.



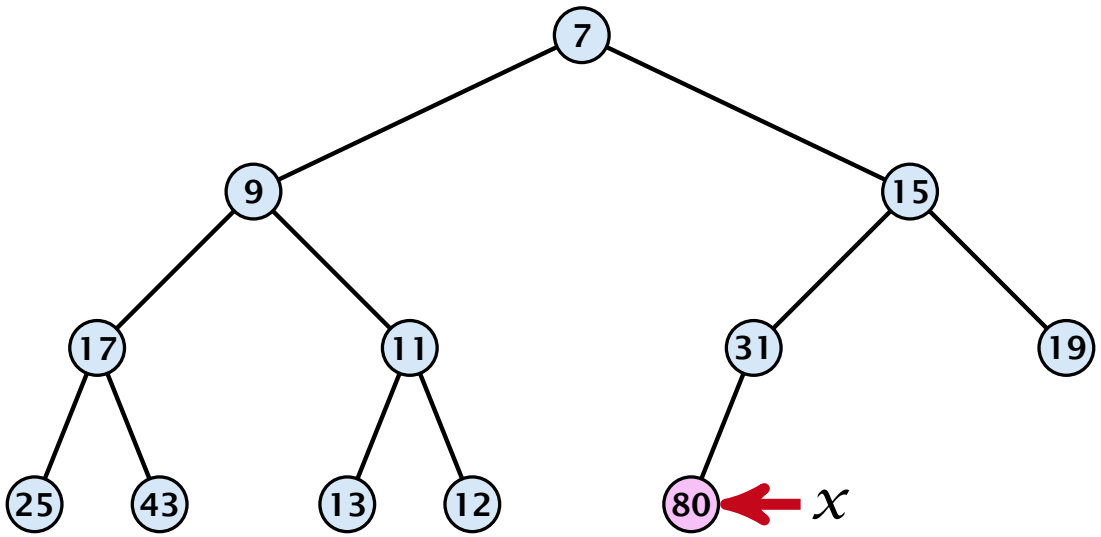
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Berechne Vorgänger von x (letztes Element wenn x gelöscht wird) in Zeit $\mathcal{O}(\log n)$.

  gehe aufwärts; stoppe nach der ersten Benutzung einer rechten Kante; gehe links; gehe rechts bis zu einem Blatt.

  wenn man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur nocht rechte Kanten.



7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Wir können Nachfolger von x (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

falls man im aufwärts-Teil auf die Wurzel trifft nimm einfach danach nur noch linke Kanten.

Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

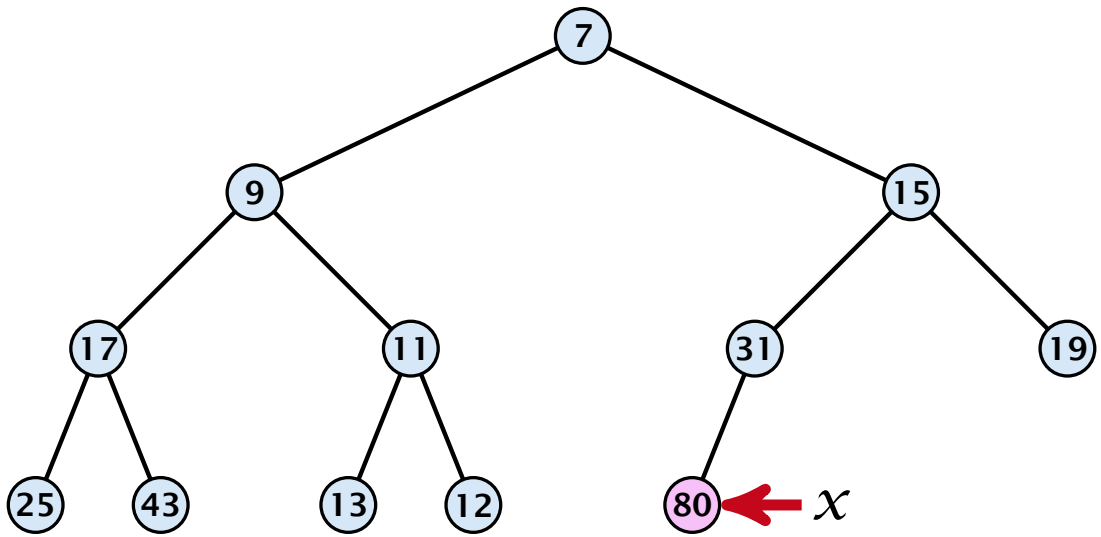
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Wir können Nachfolger von x (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

falls man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch linke Kanten.



Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

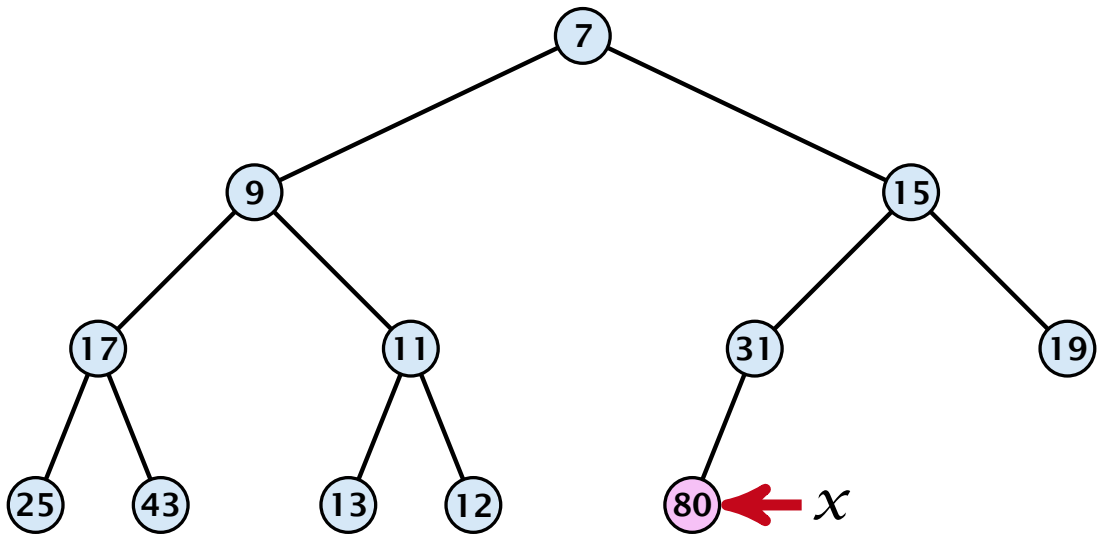
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Wir können Nachfolger von x (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

  gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

  falls man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch linke Kanten.



Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

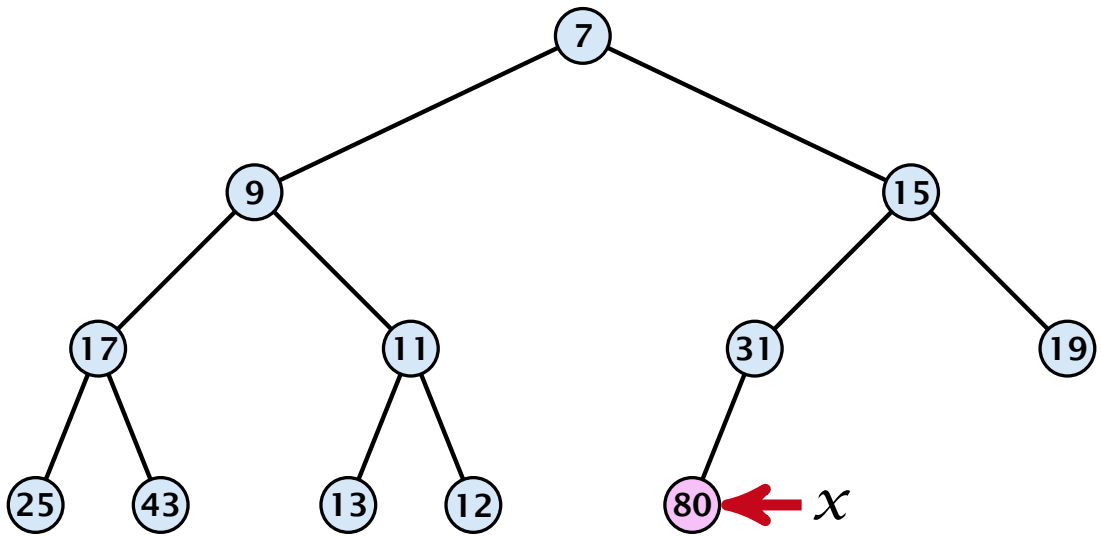
7.3 Priority Queues

Verwalte Zeiger auf letztes Element x.

- Wir können Nachfolger von x (letztes Element wenn ein Element eingefügt wird) in Zeit $\mathcal{O}(\log n)$ berechnen.

  gehe aufwärts; stoppe nach Benutzung einer linken Kante; gehe rechts; nimm linke Kanten.

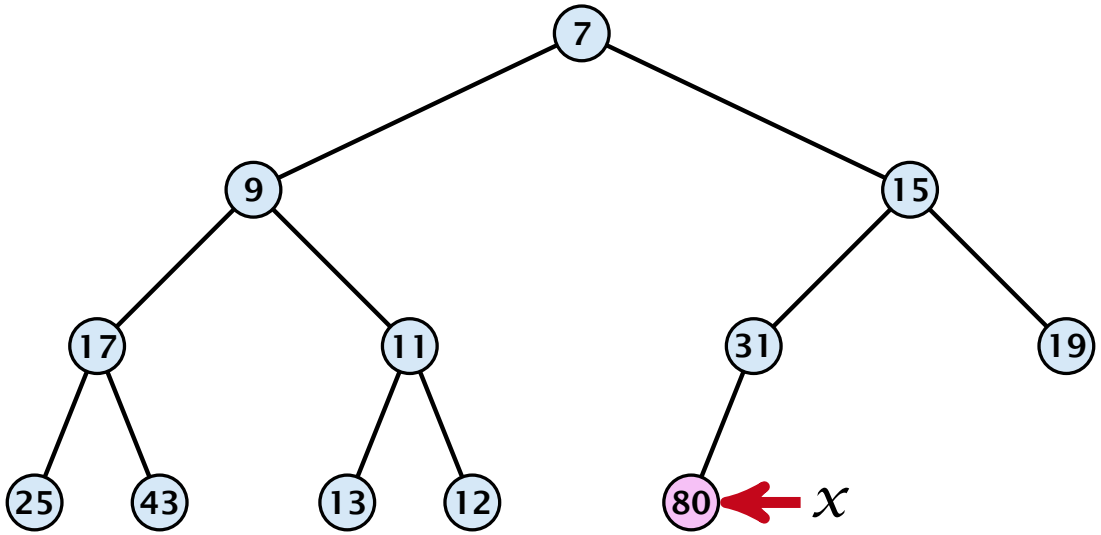
  falls man im aufwärts-Teil auf die Wurzel trifft nimmt man danach nur noch linke Kanten.



Eigentlich wird hier nicht der Nachfolger berechnet sondern die Stelle an der man ein Element einfügen würde. Der Abwärtsteil endet an einem NULL-pointer, d.h. man möchte eine linke oder rechte Kante nehmen, aber das dazugehörige Kind existiert nicht. An dieser Stelle würde man ein Element einfügen.

Einfügen

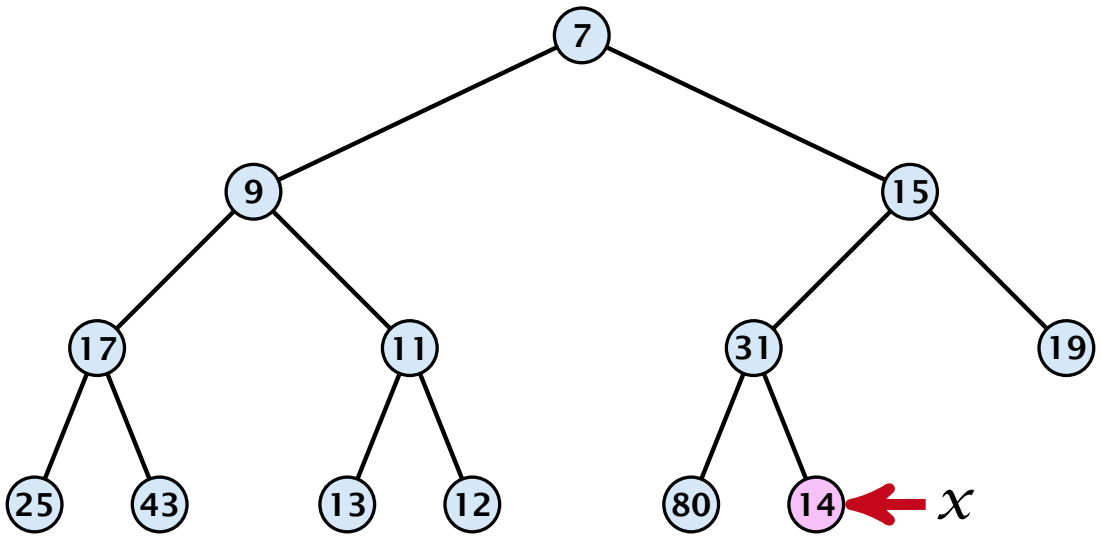
1. Füge Element am Nachfolger von x ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.



Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

Einfügen

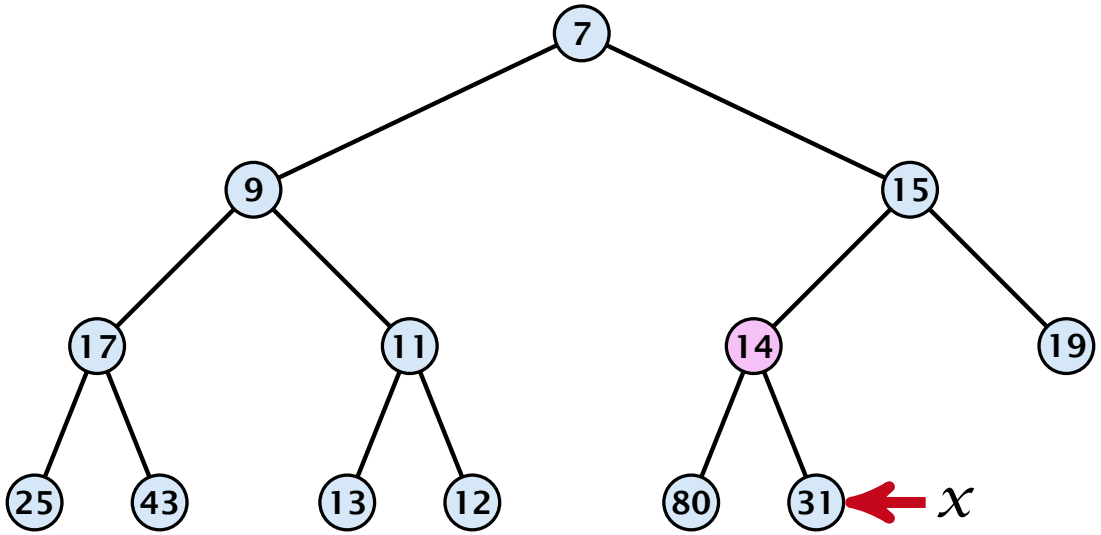
1. Füge Element am Nachfolger von x ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.



Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

Einfügen

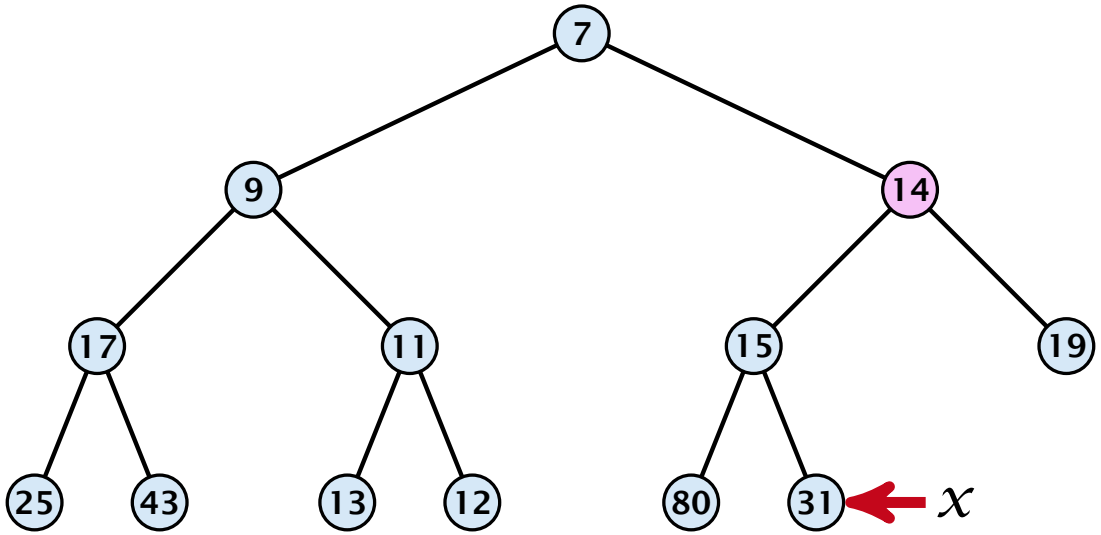
1. Füge Element am Nachfolger von x ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.



Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

Einfügen

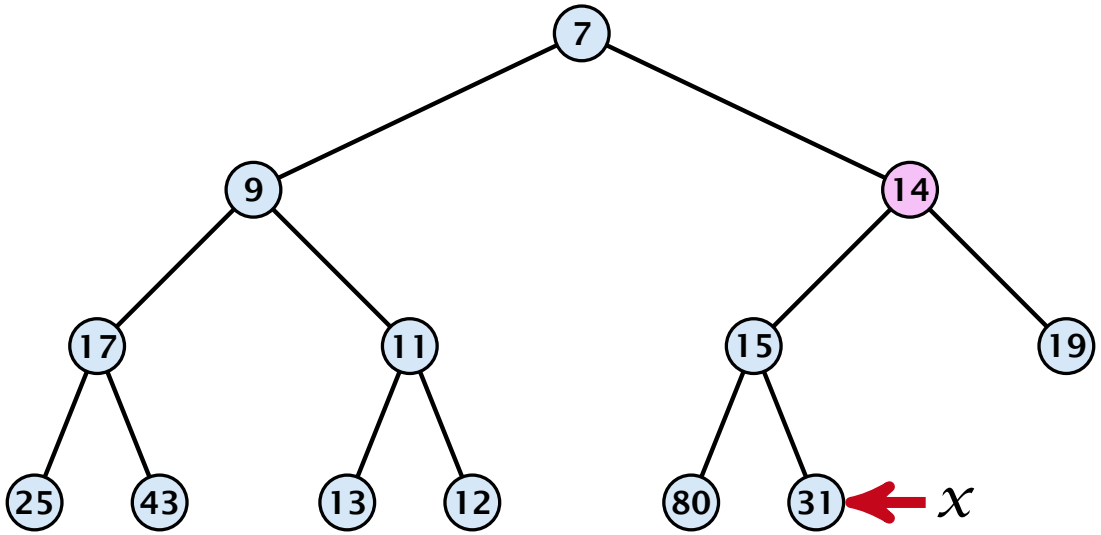
1. Füge Element am Nachfolger von x ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.



Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

Einfügen

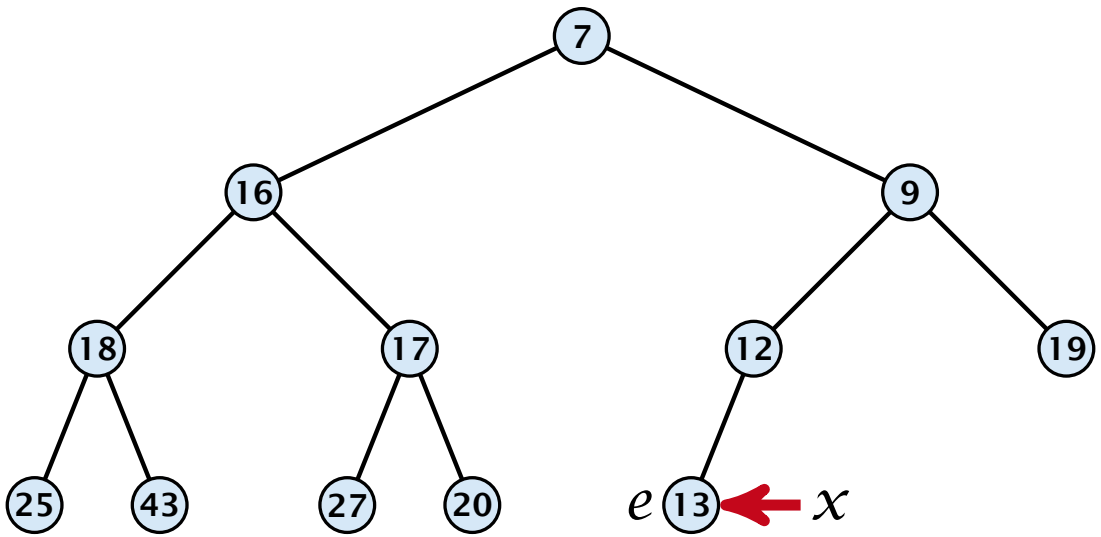
1. Füge Element am Nachfolger von x ein.
2. Tausche Element mit Elternknoten bis die Heapeigenschaft erfüllt ist.



Swaps können durch Pointermanipulation oder Datenaustausch realisiert werden. Die erste Variante ergibt eine addressierbare Priority Queue.

Löschen

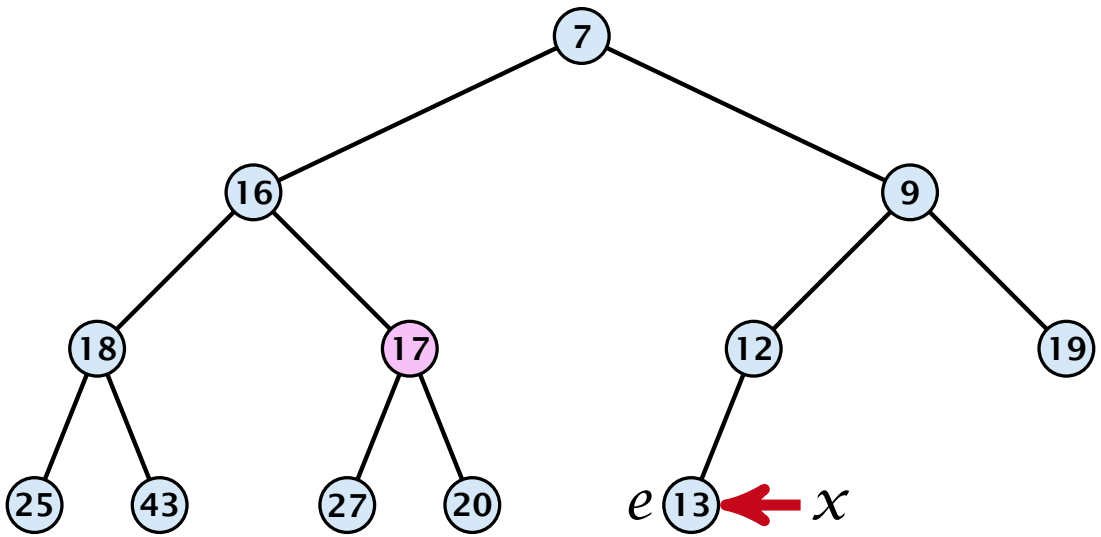
1. Vertausche zu löschenes Element mit dem letzen Element e.
2. Stelle die Heapeigenschaft für Element e wieder her.



An der neuen Position wandert e entweder auf oder abwärts (aber nicht beides).

Löschen

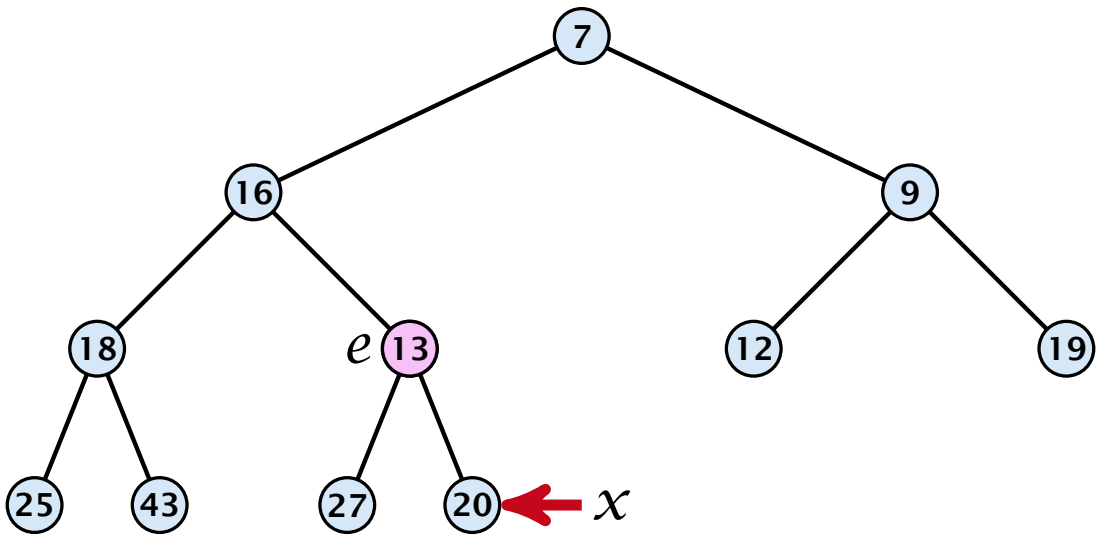
1. Vertausche zu löschenes Element mit dem letzen Element e.
2. Stelle die Heapeigenschaft für Element e wieder her.



An der neuen Position wandert e entweder auf oder abwärts (aber nicht beides).

Löschen

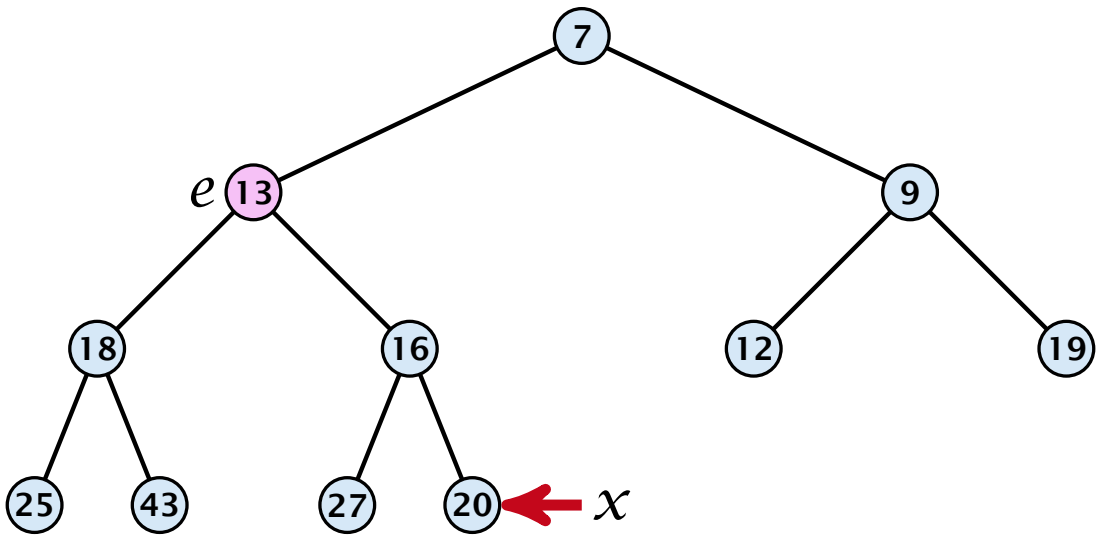
1. Vertausche zu löschenes Element mit dem letzen Element e.
2. Stelle die Heapeigenschaft für Element e wieder her.



An der neuen Position wandert e entweder auf oder abwärts (aber nicht beides).

Löschen

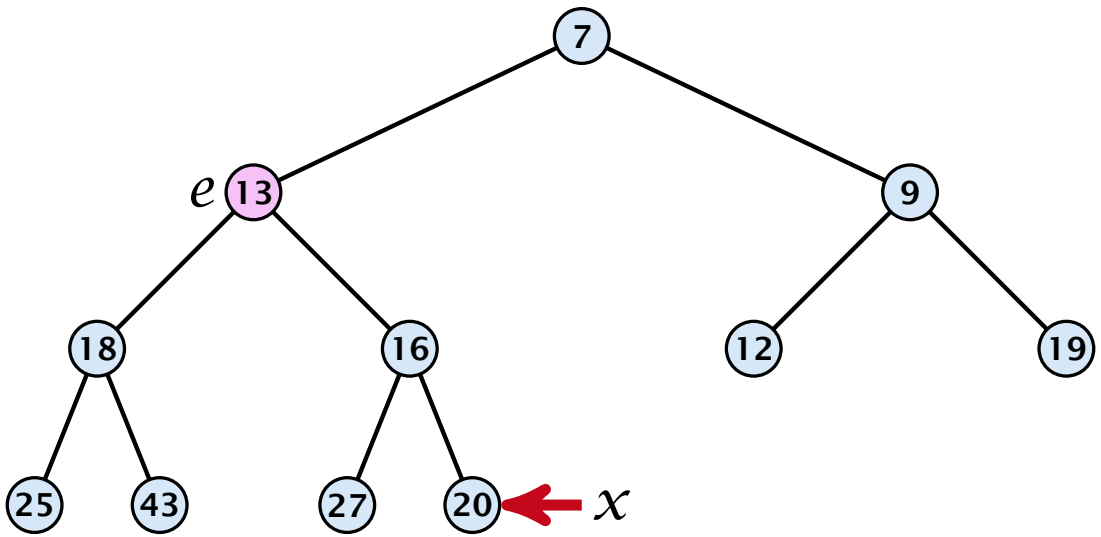
1. Vertausche zu löschenes Element mit dem letzen Element e.
2. Stelle die Heapeigenschaft für Element e wieder her.



An der neuen Position wandert e entweder auf oder abwärts (aber nicht beides).

Löschen

1. Vertausche zu löschenes Element mit dem letzen Element e.
2. Stelle die Heapeigenschaft für Element e wieder her.



An der neuen Position wandert e entweder auf oder abwärts (aber nicht beides).

Binäre Heaps

Operationen:

- **minimum():** Gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** Überprüfe ob Wurzelzeiger NULL. Zeit $\mathcal{O}(1)$.
- **insert(k):** füge bei Nachfolger von x ein. bubble up. Zeit $\mathcal{O}(\log n)$.
- **delete(h):** tausche mit x; bubble up **or** sift-down. Zeit $\mathcal{O}(\log n)$.

Binäre Heaps

```
1 void bubbleUp(TreeNode* v) {
2     while (v->parent && v->parent->val > v->val)
3         swap(v,v->parent);
4 }
```

Vertausche mit Elternknoten solange dein Wert kleiner als Wert des Elternknotens.

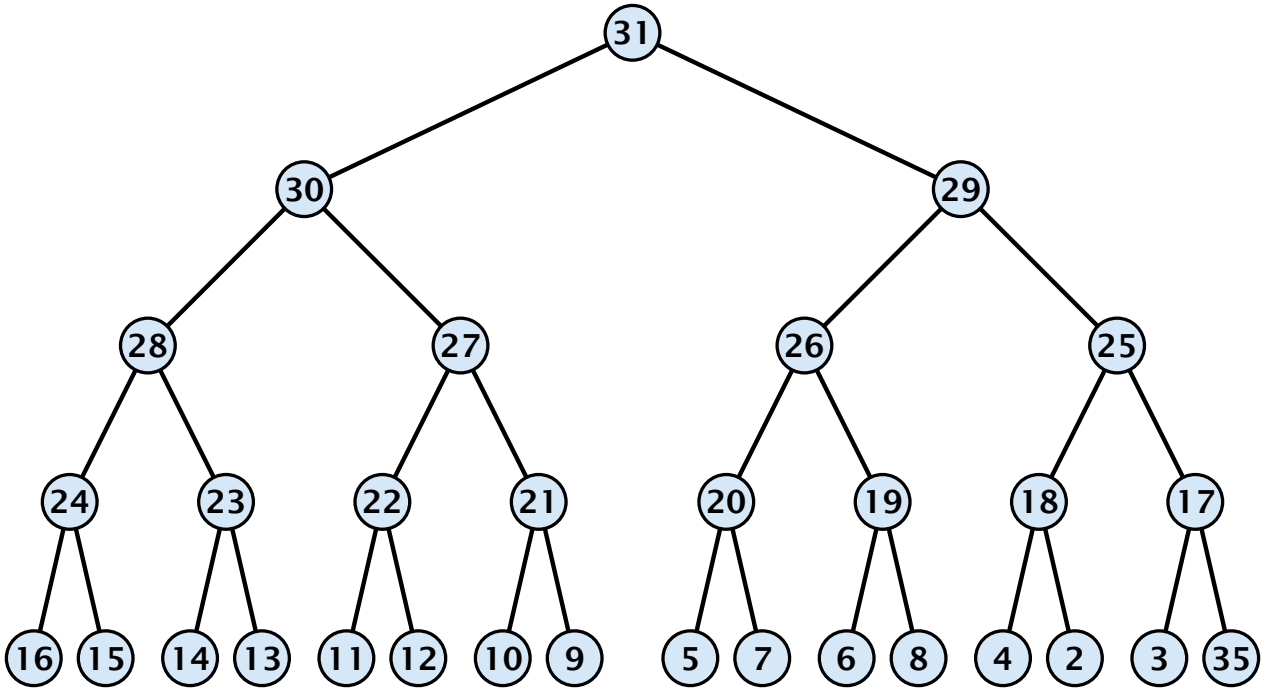
Binäre Heaps

```
1  void siftDown(TreeNode* v) {
2      TreeNode* m = NULL;
3      while (true) {
4          int minValue = v->val;
5          if (v->left && minValue > v->left->val) {
6              minValue = v->left->val;
7              m = v->left;
8          }
9          if (v->right && minValue > v->right->val) {
10             minValue = v->right->val;
11             m = v->right;
12         }
13         if (v->val != minValue)
14             swap(v,m);
15         else break;
16     }
17 }
```

Vertausche mit dem kleineren der Kinder, solange der Wert kleiner als dein eigener ist.

Build Heap

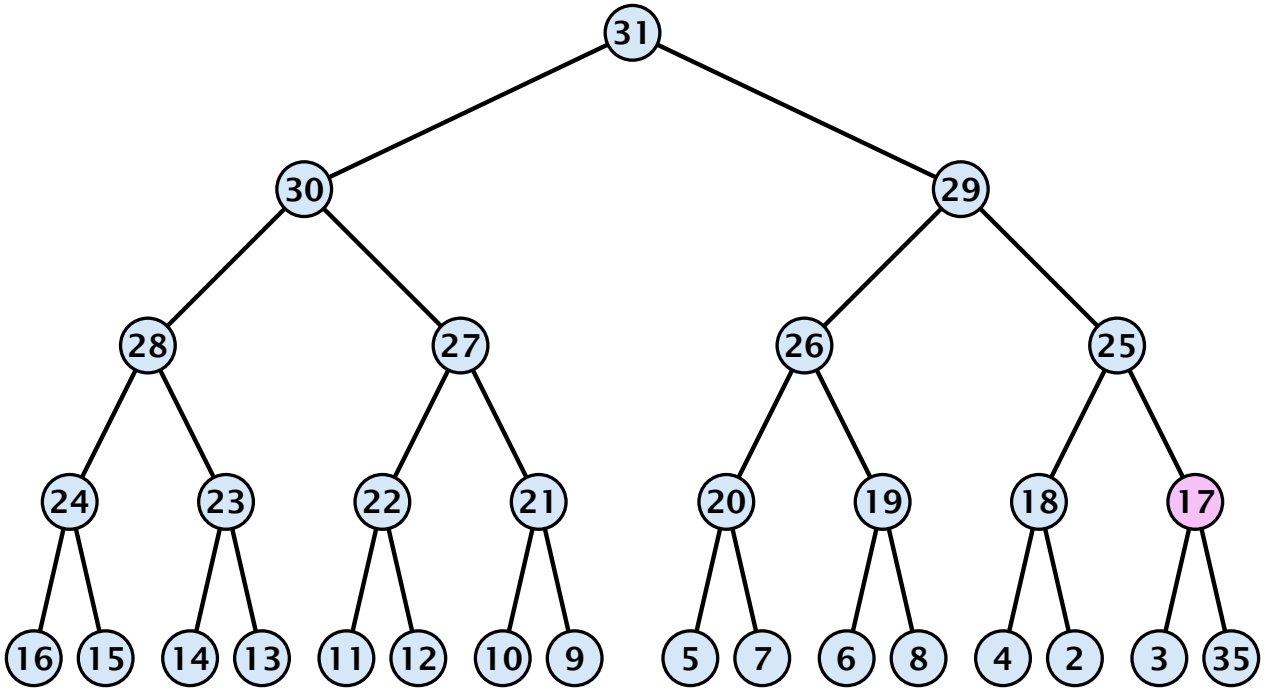
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

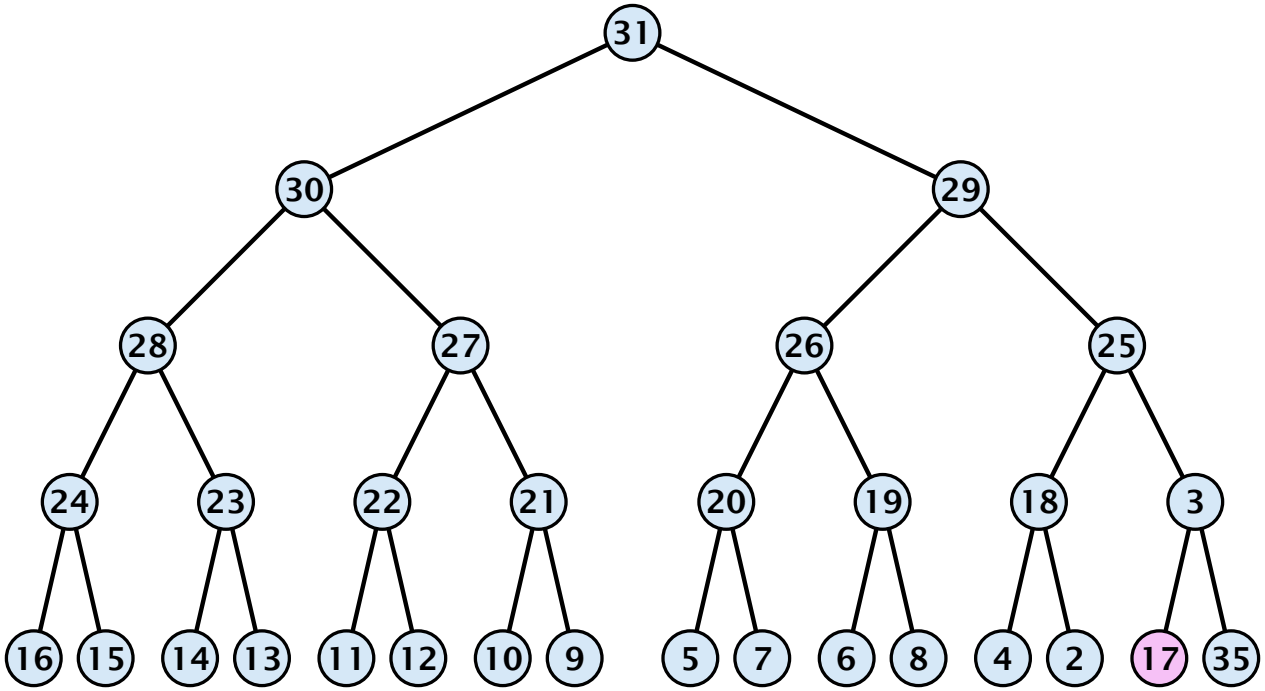
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

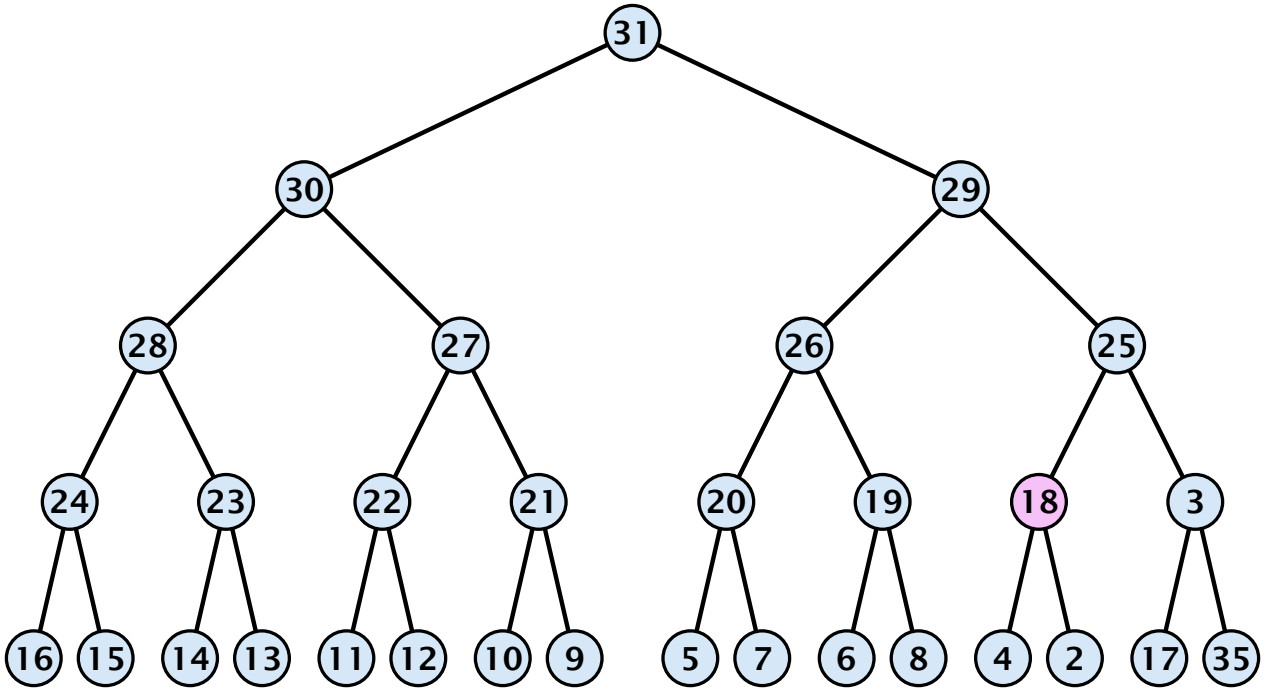
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

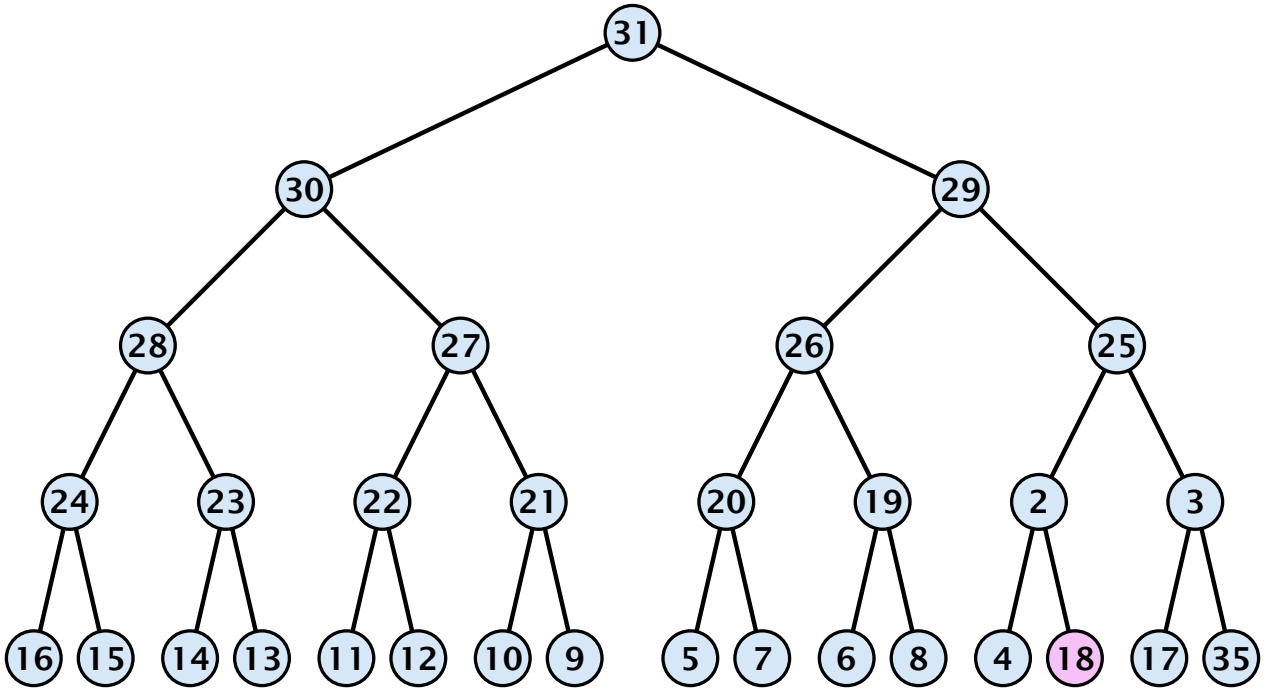
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

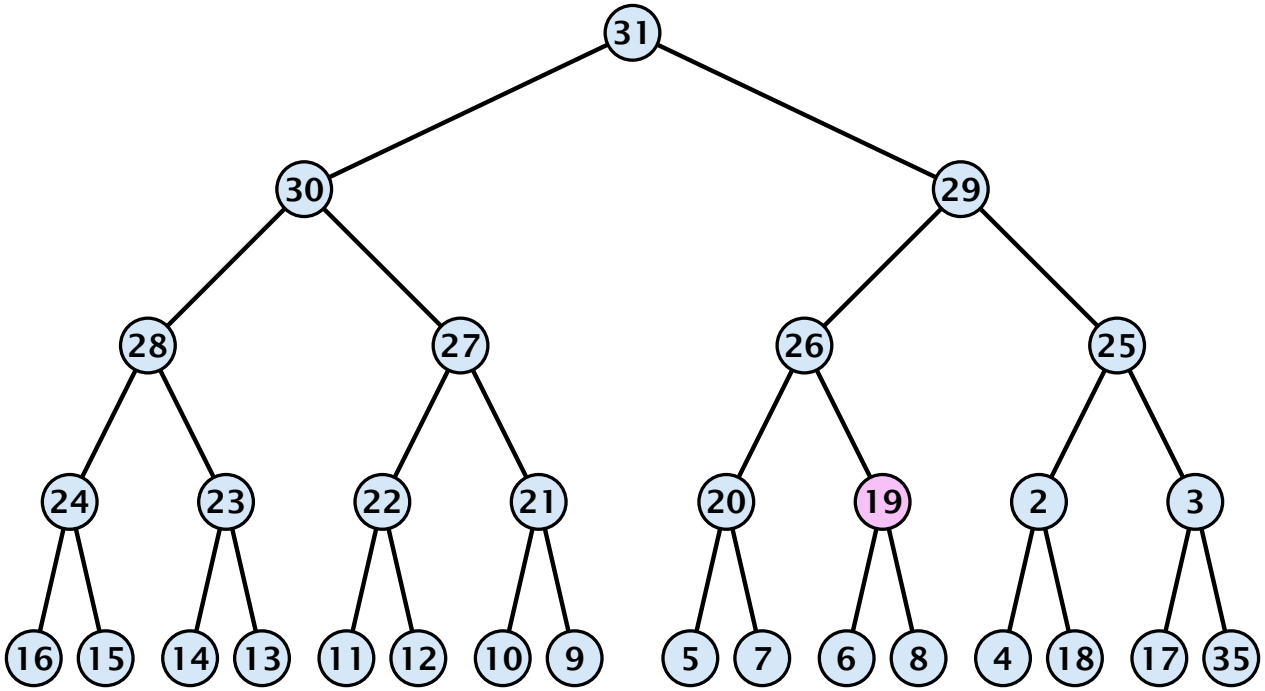
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

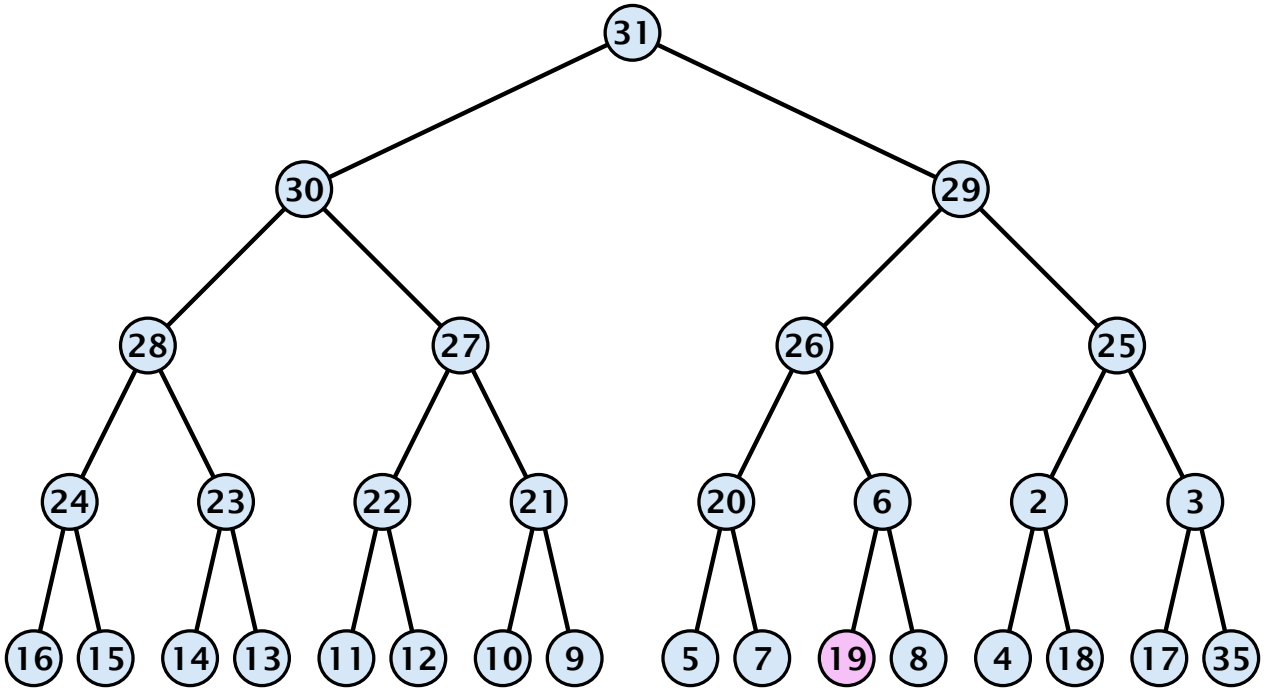
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

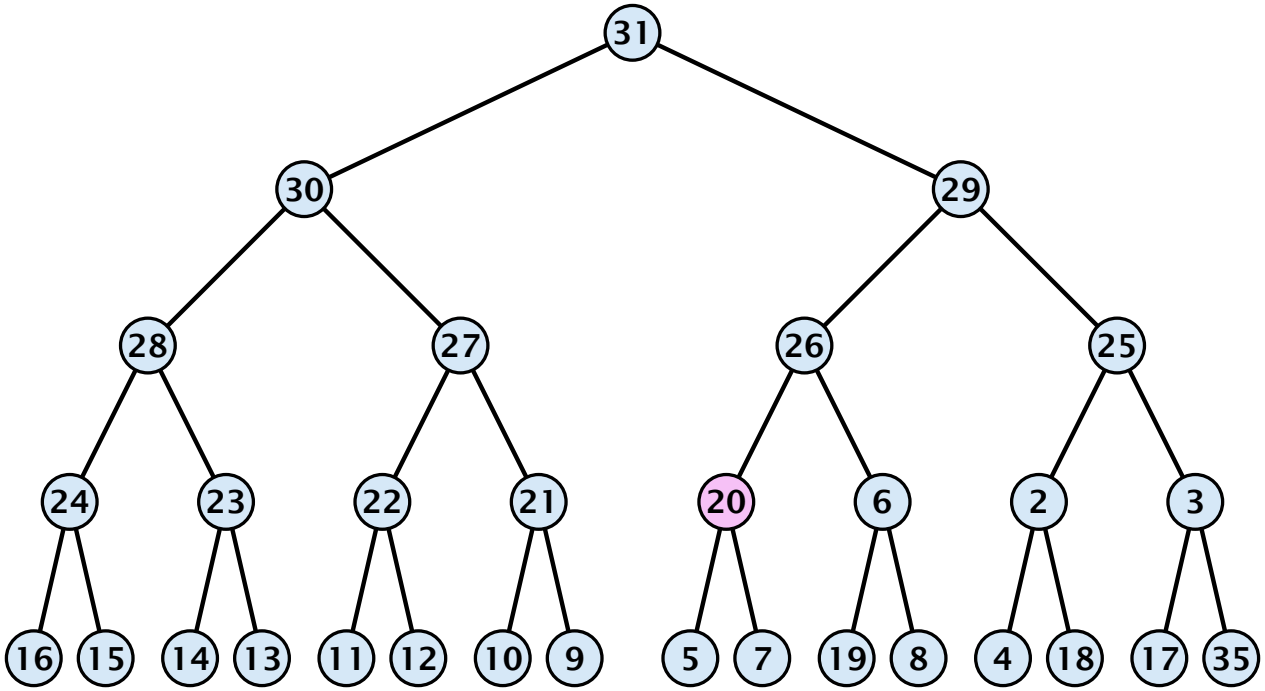
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

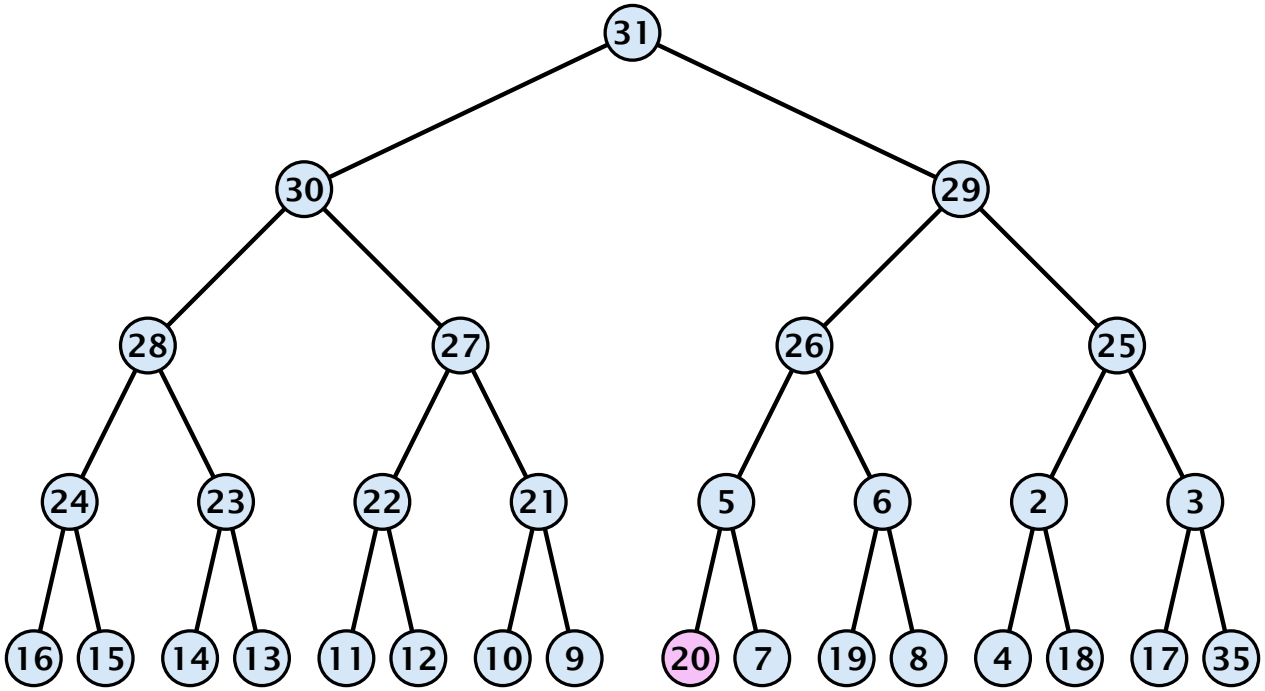
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

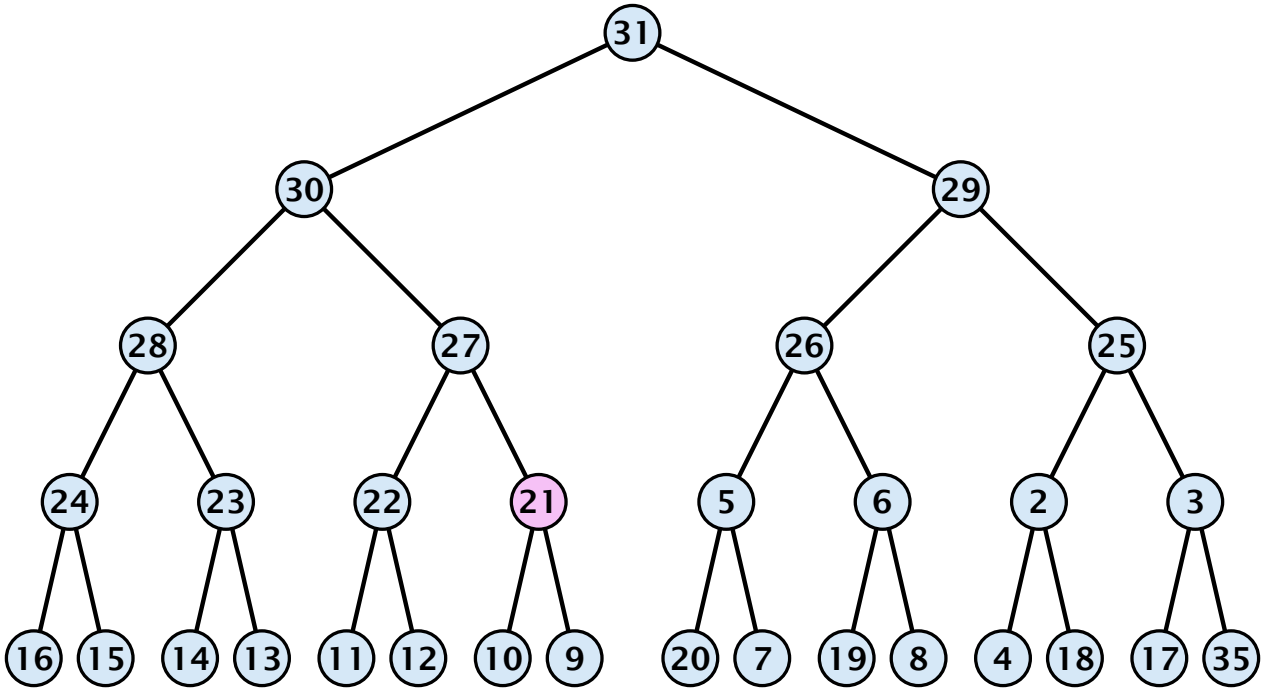
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

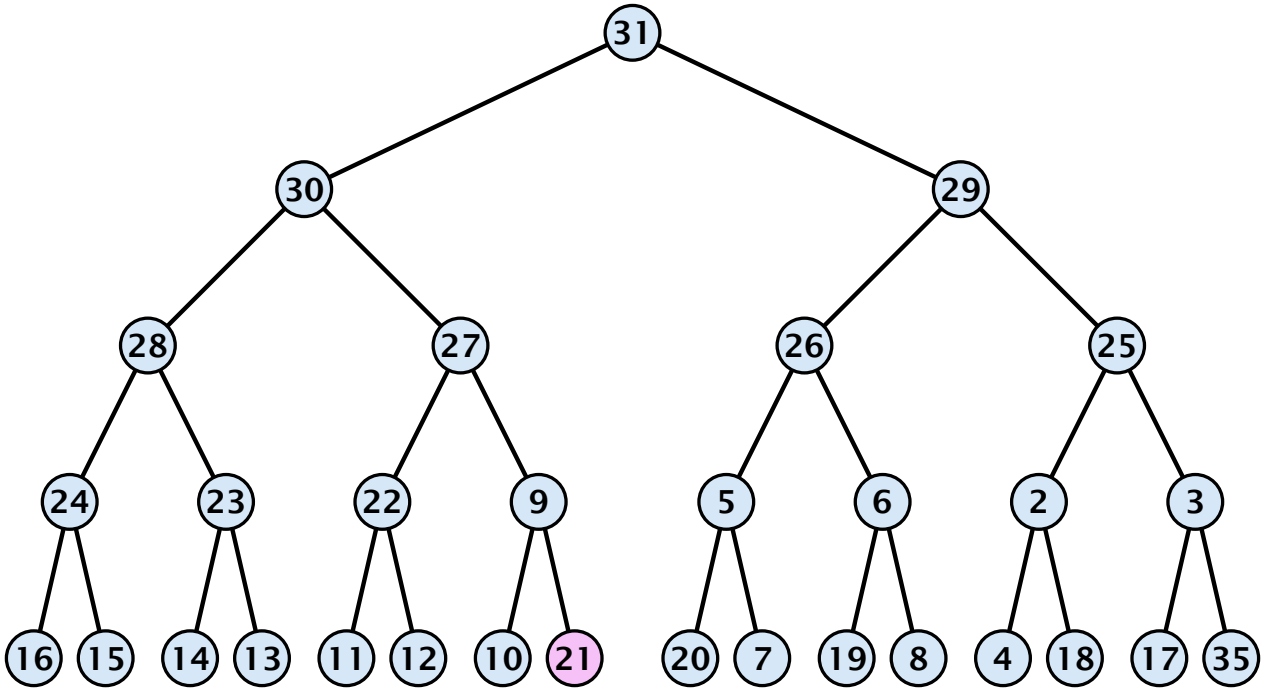
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

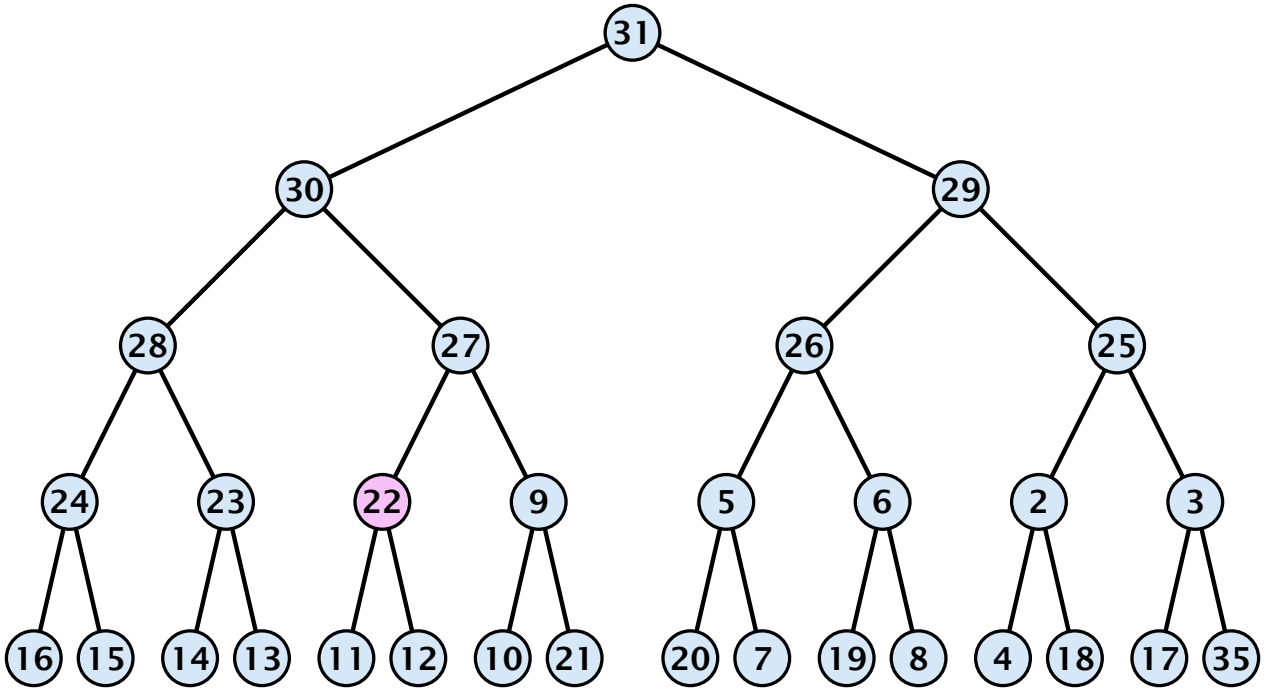
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

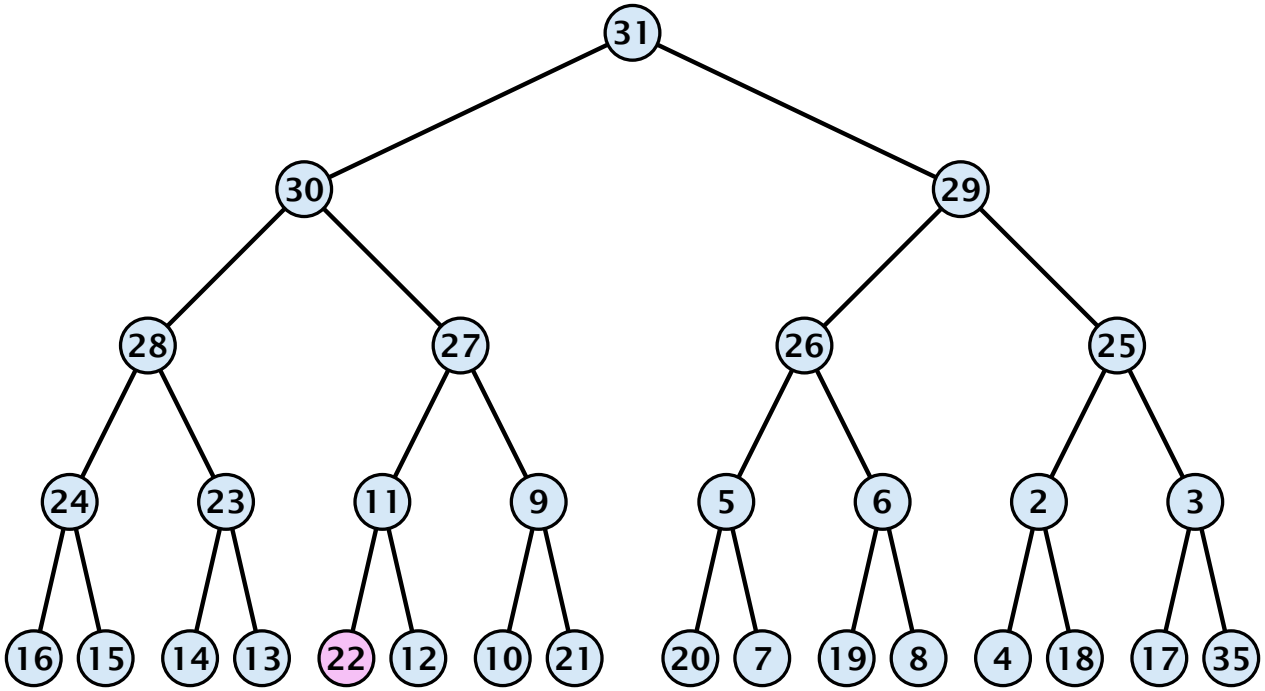
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

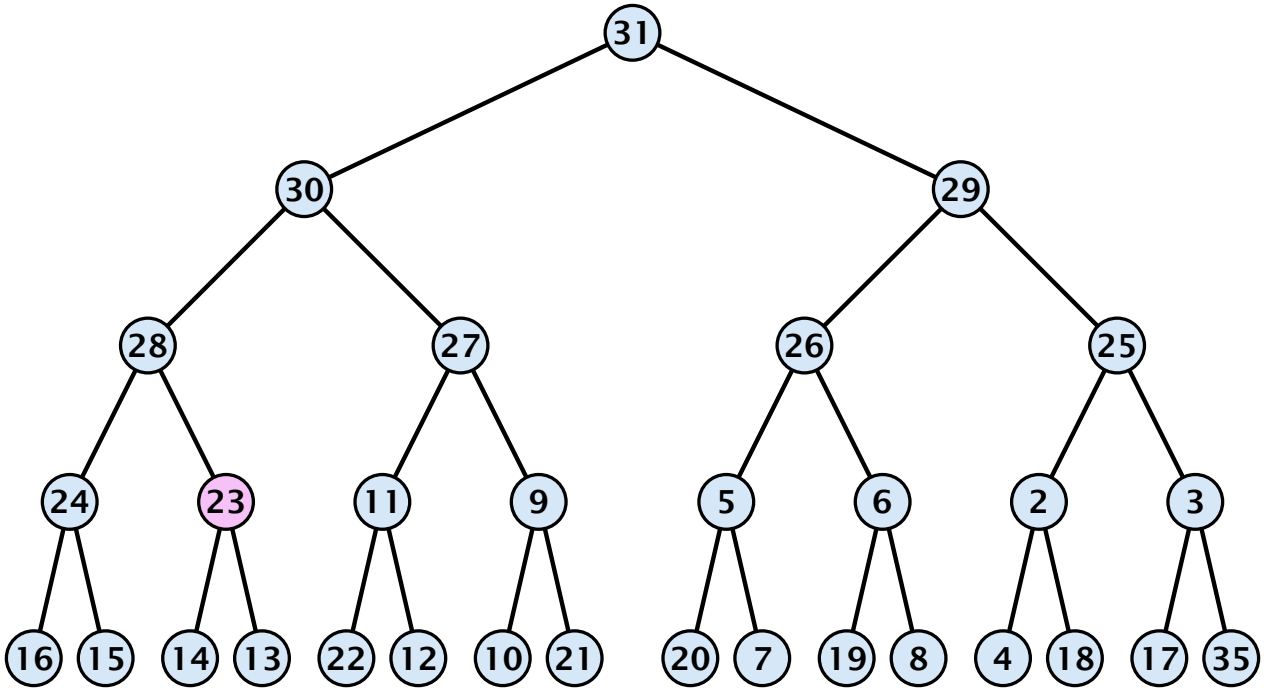
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

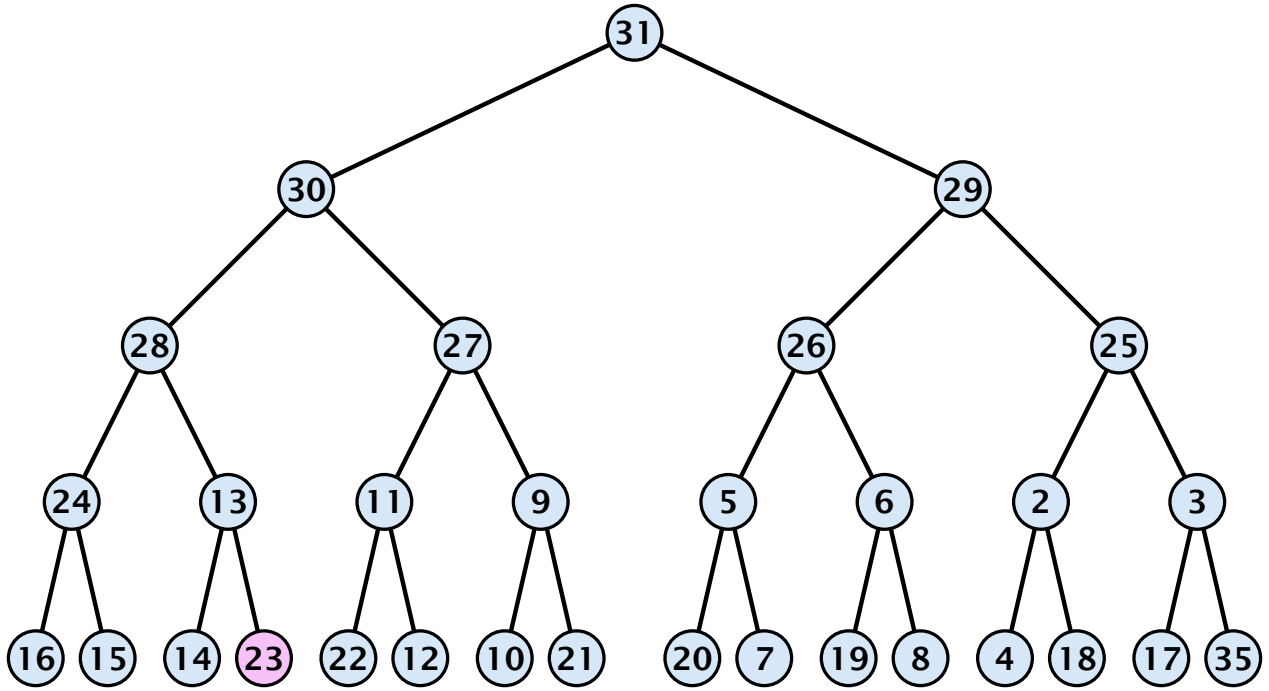
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

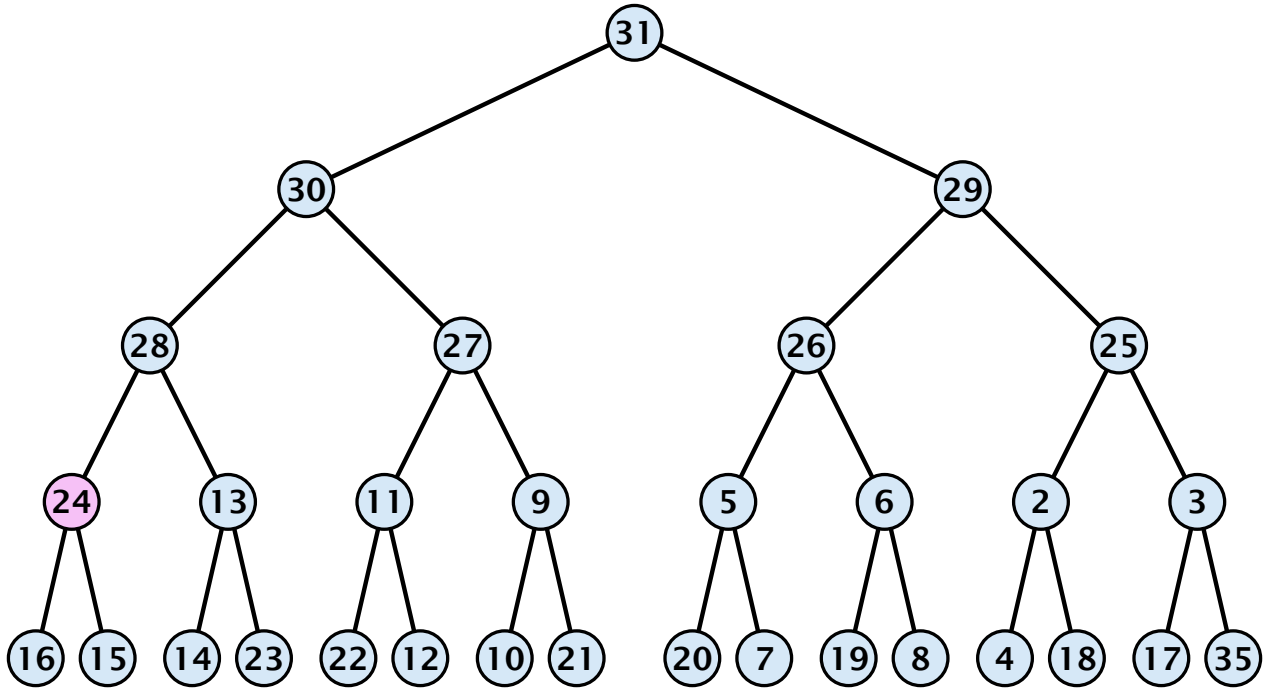
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

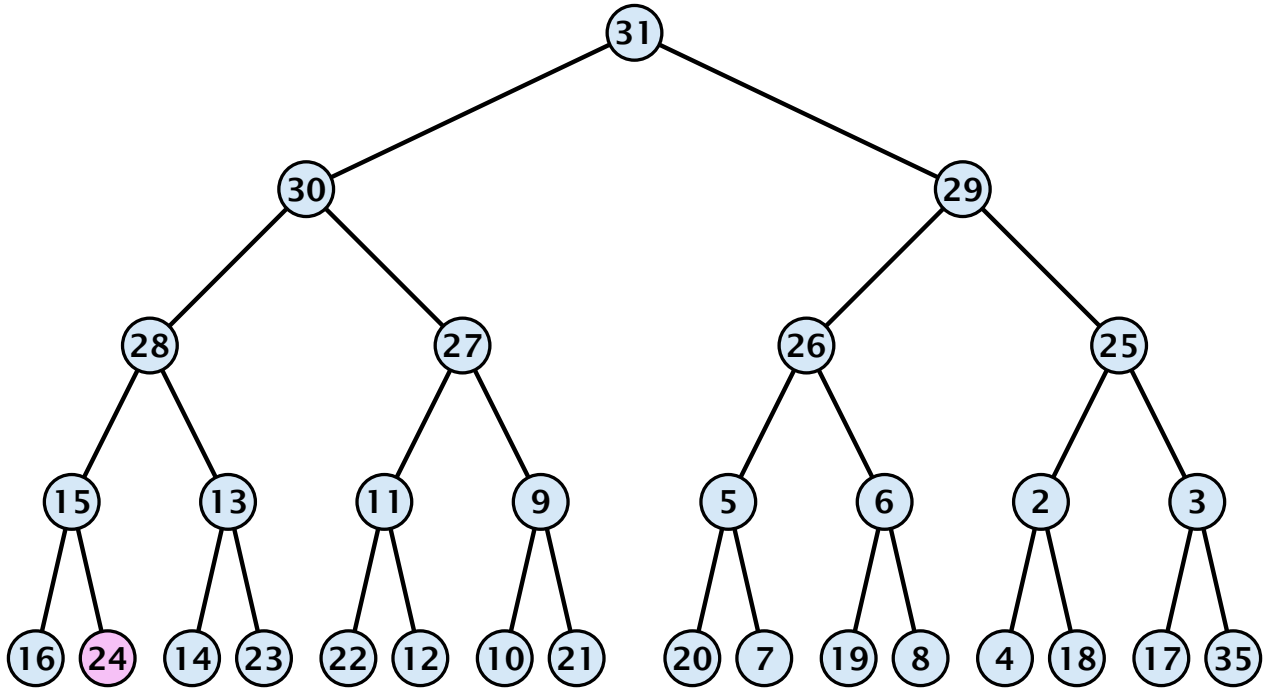
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

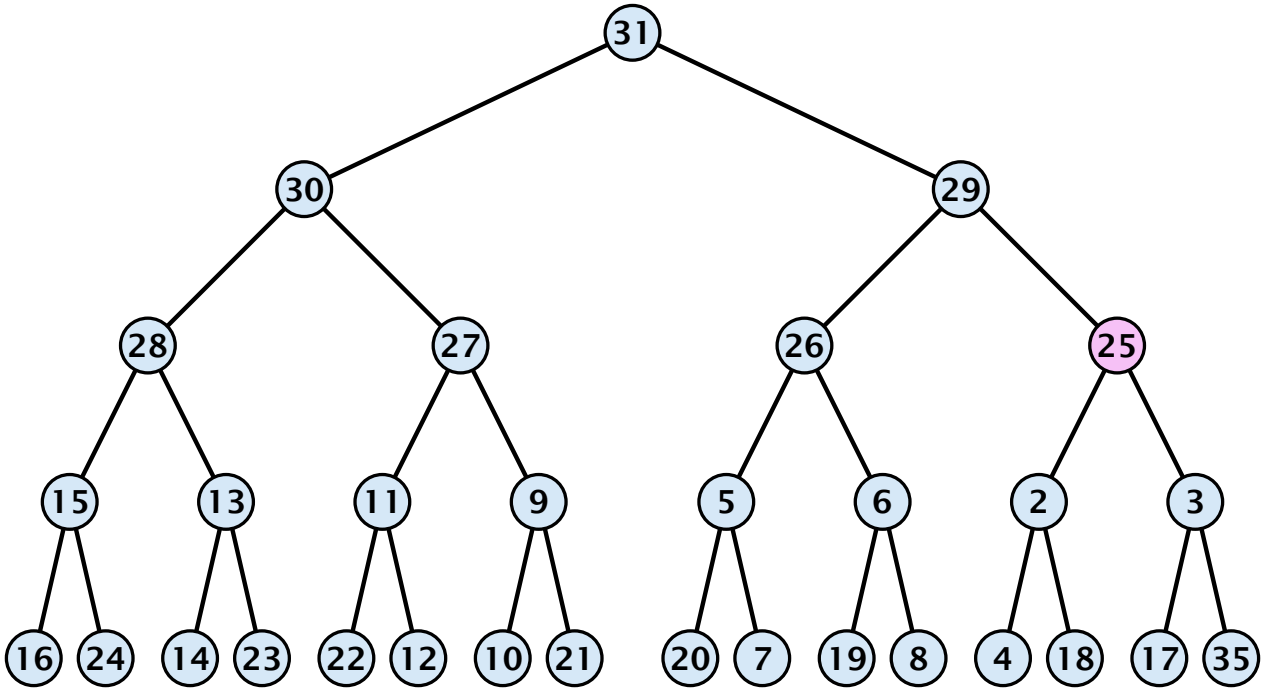
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

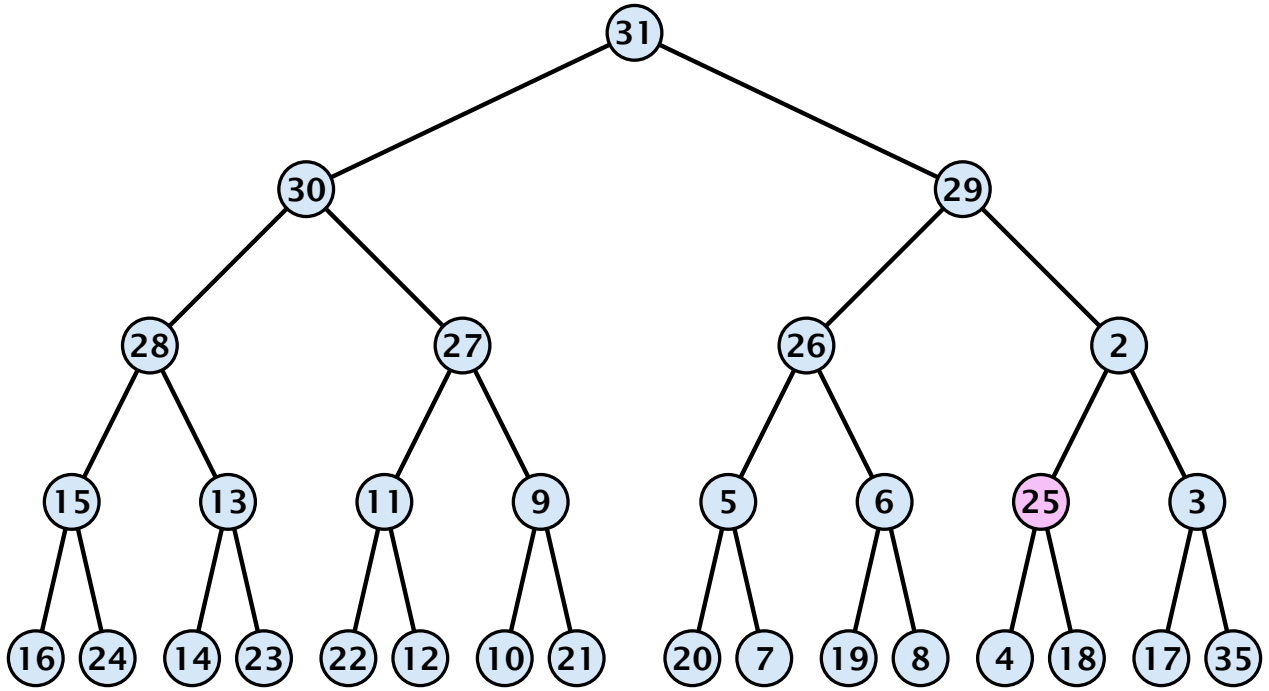
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

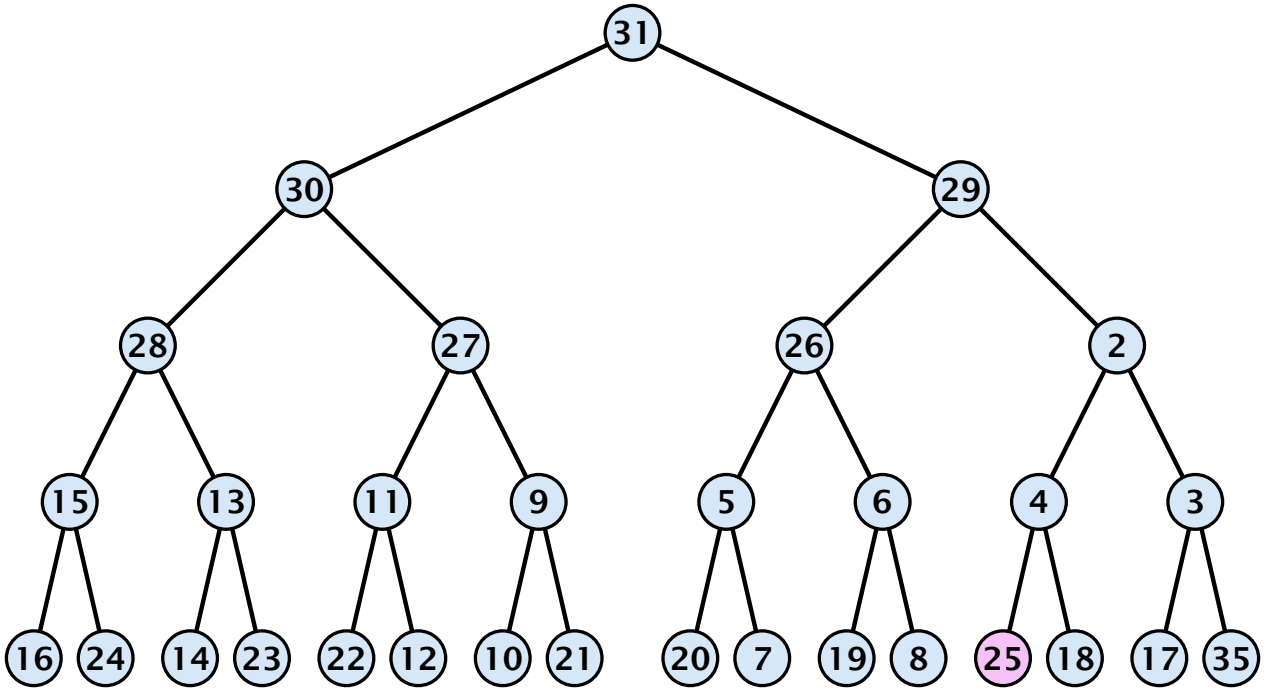
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

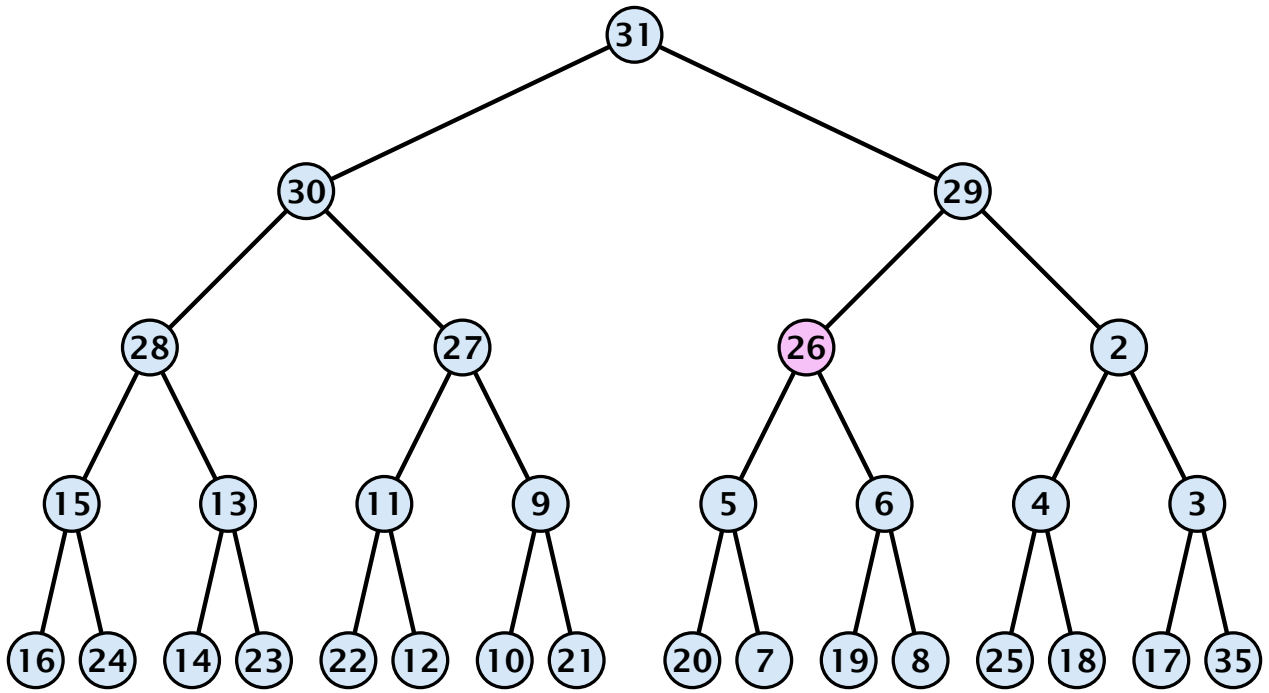
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

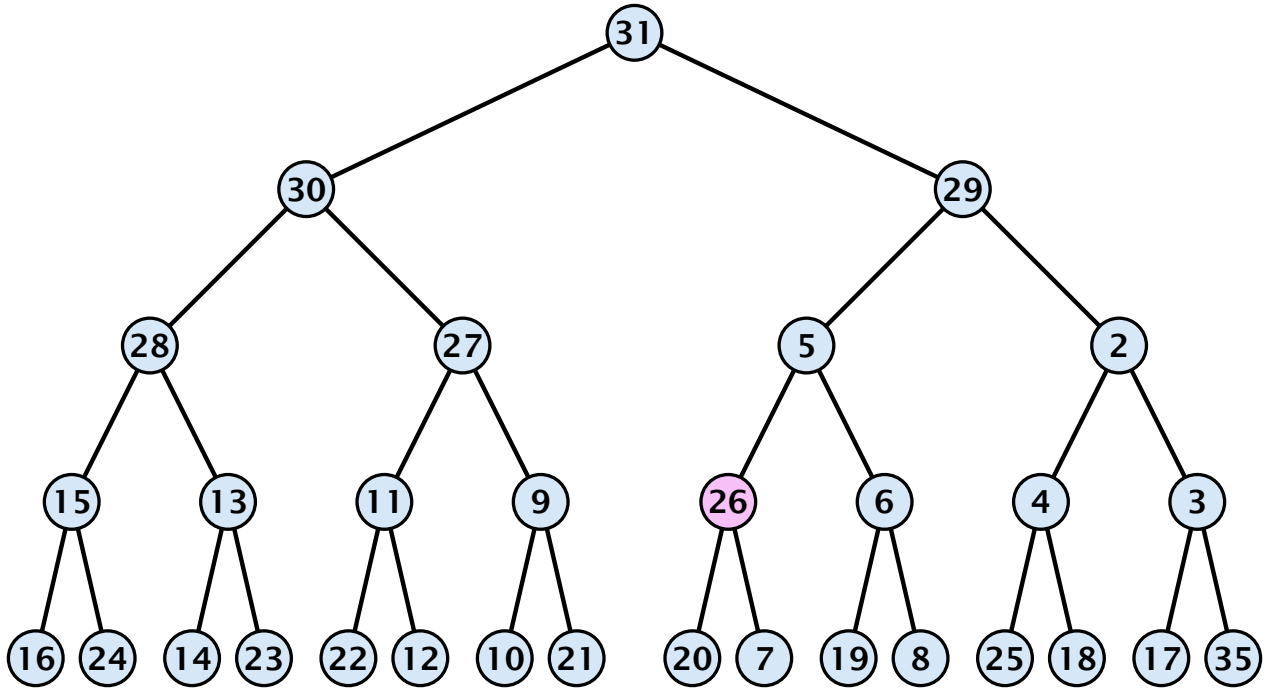
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

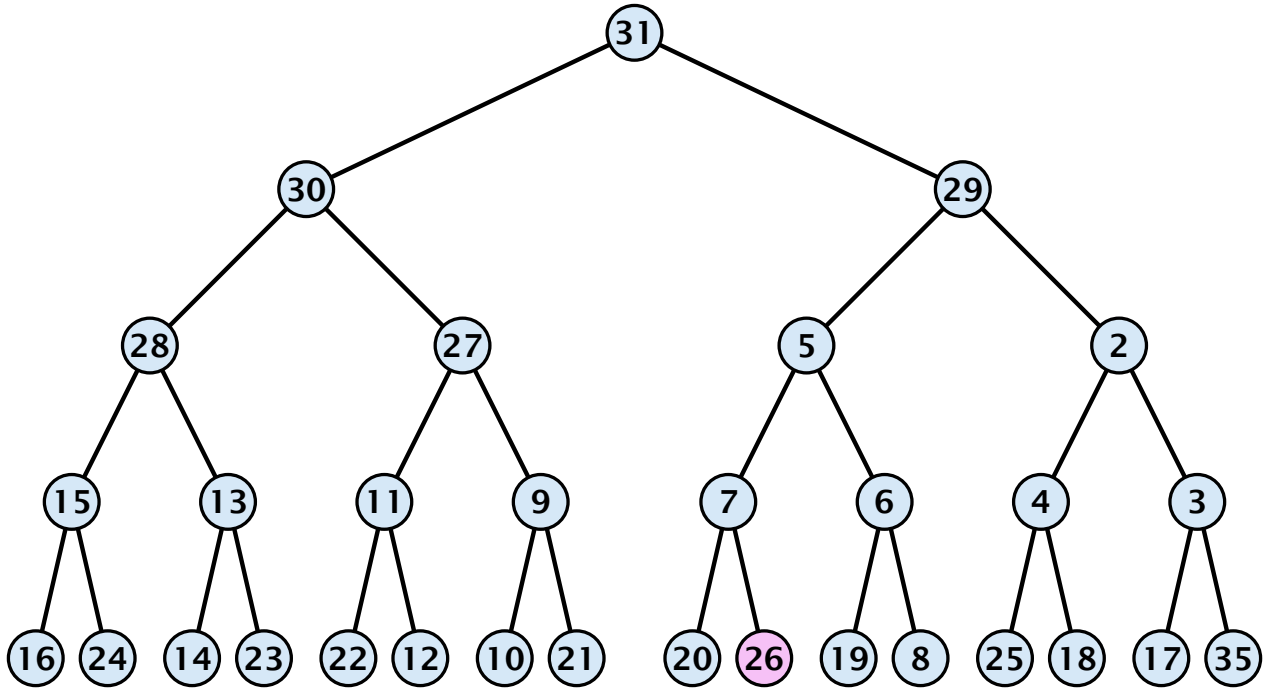
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

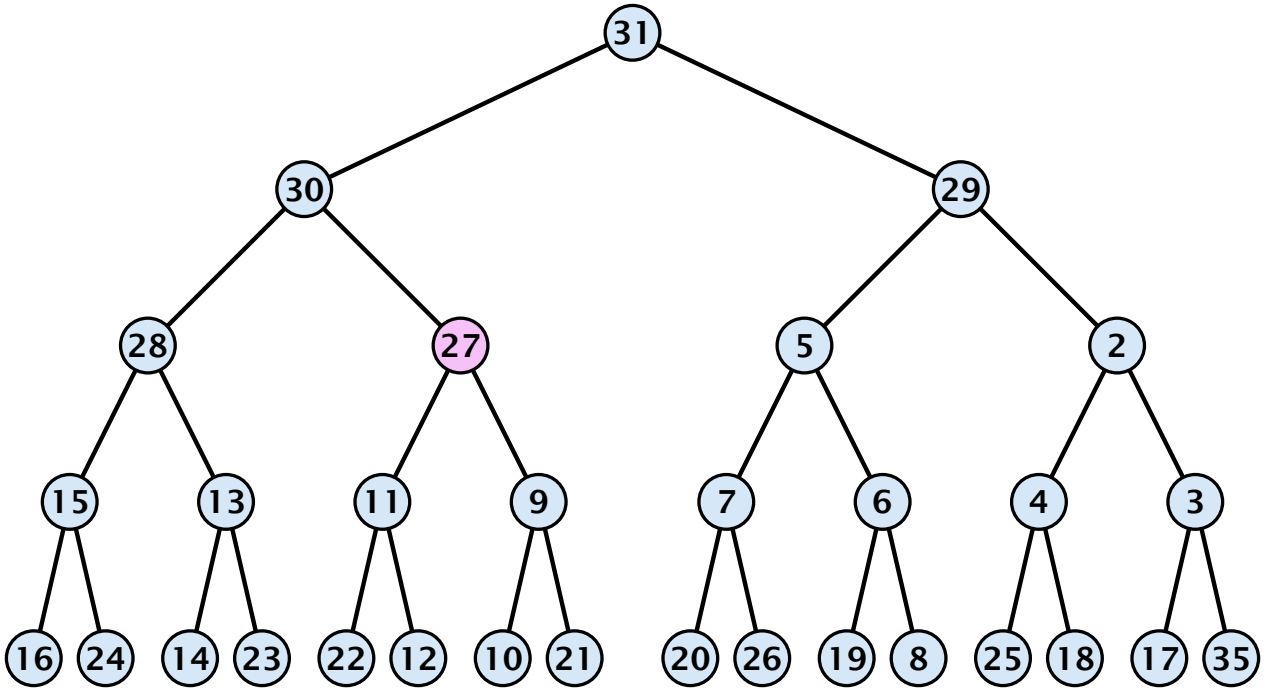
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

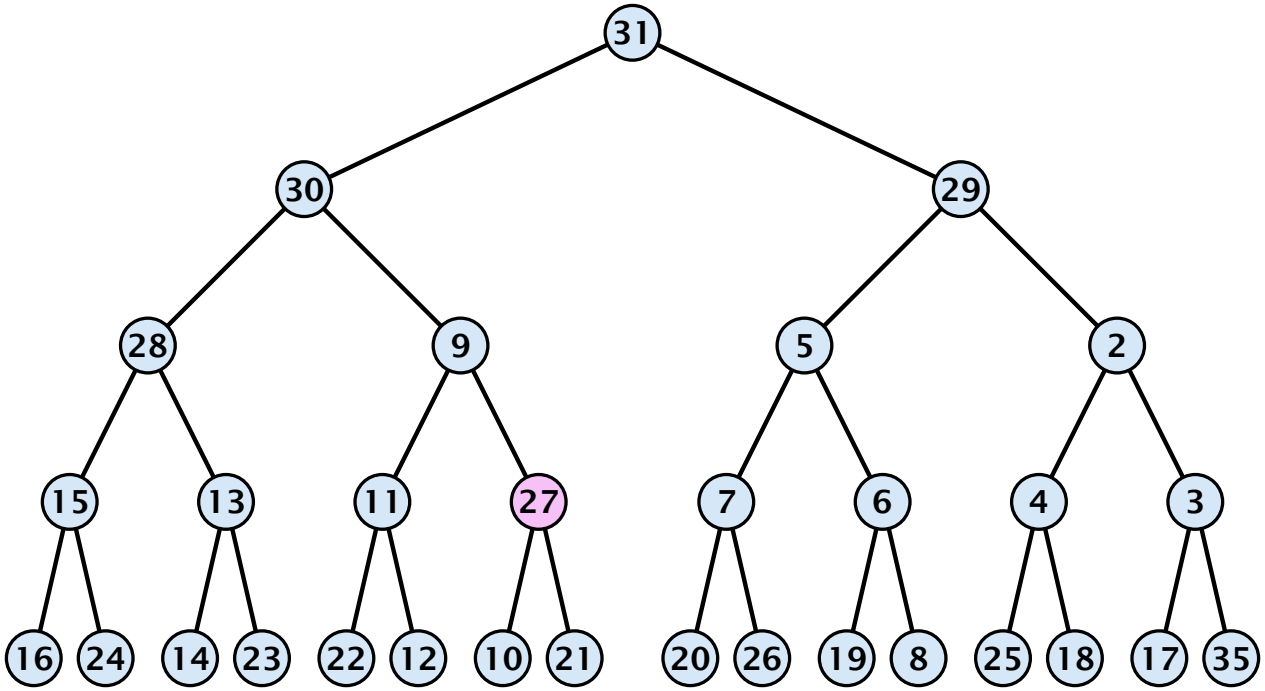
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

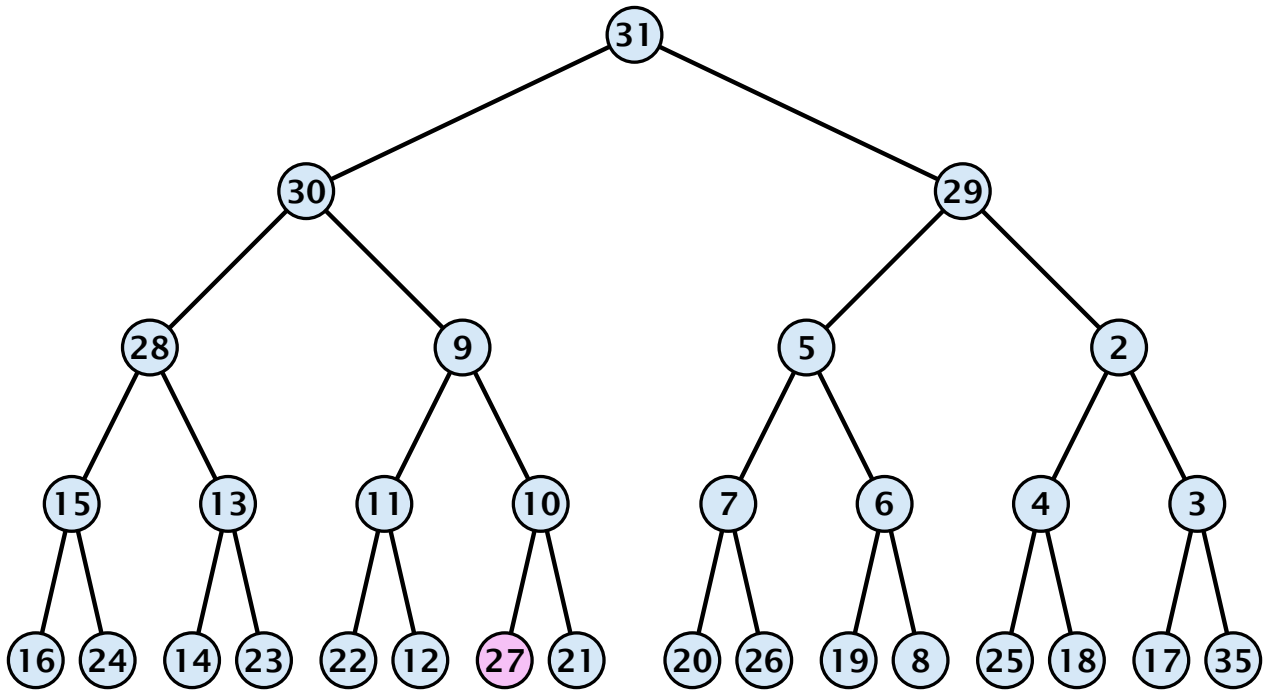
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

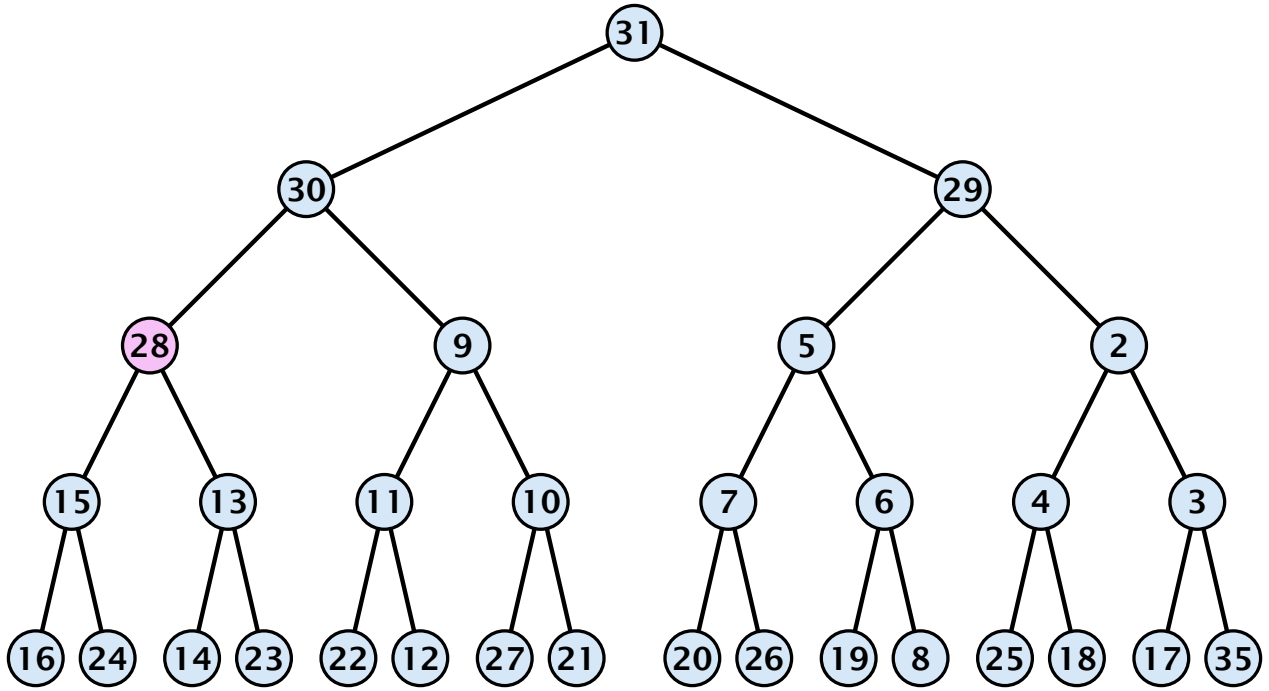
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

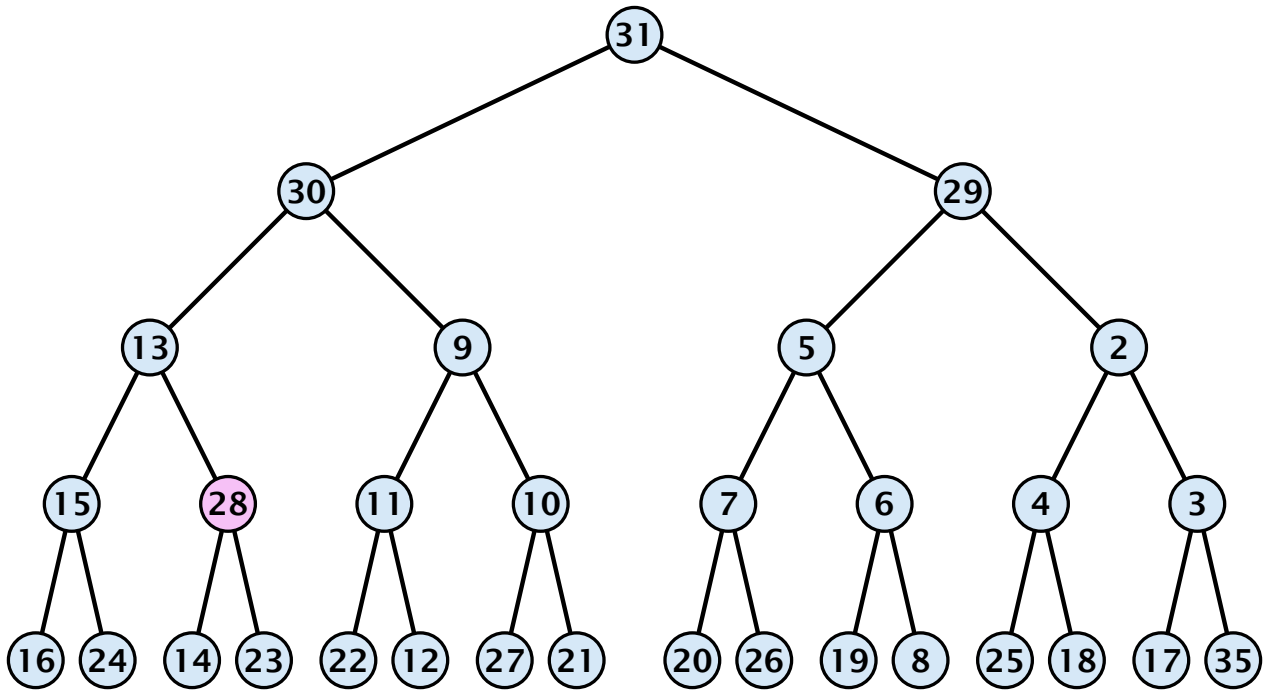
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

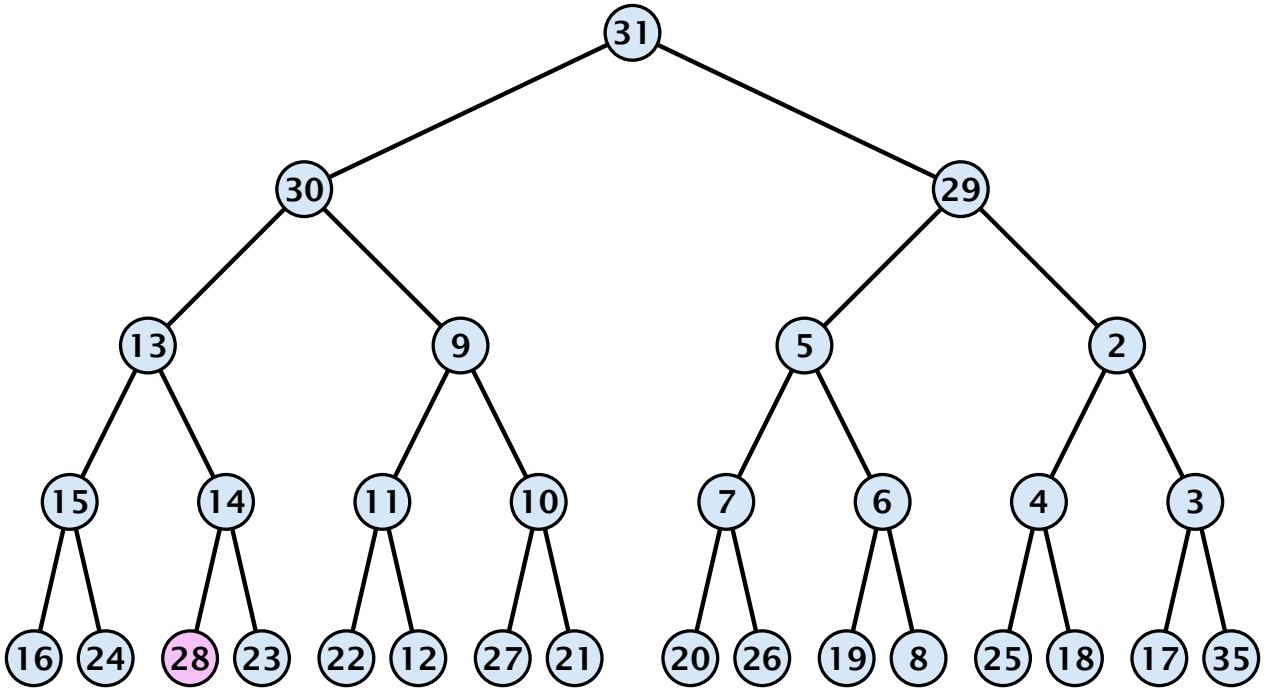
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

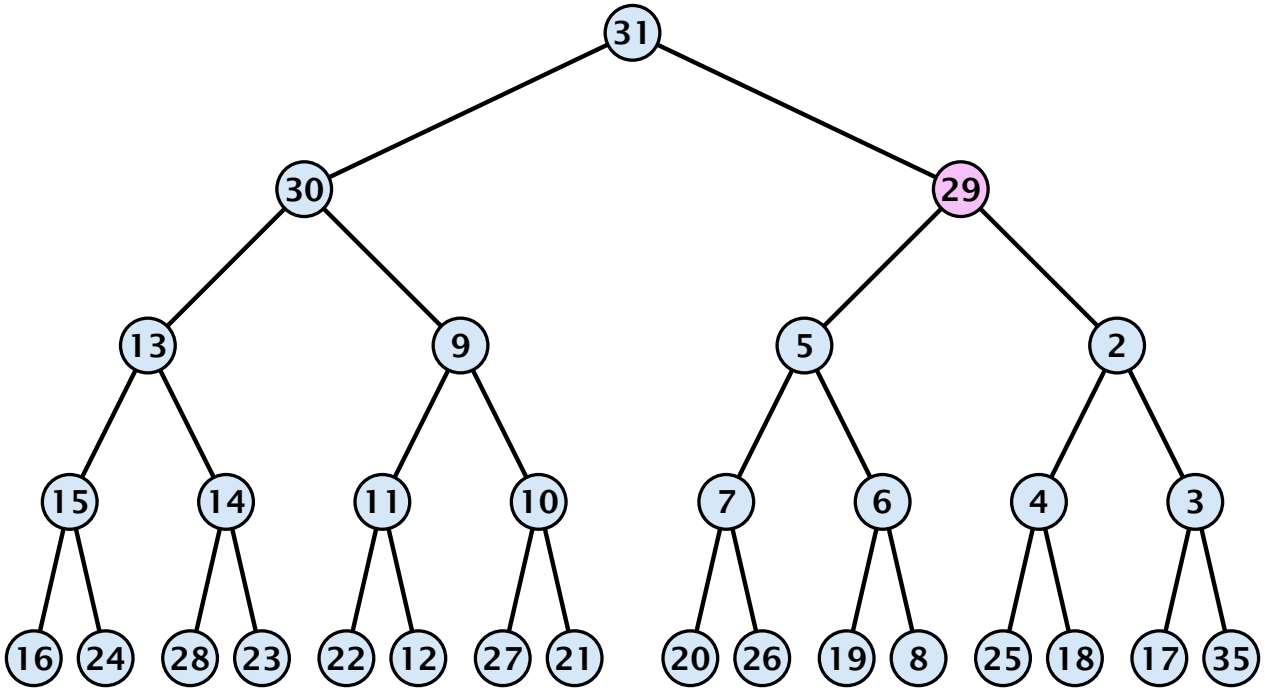
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

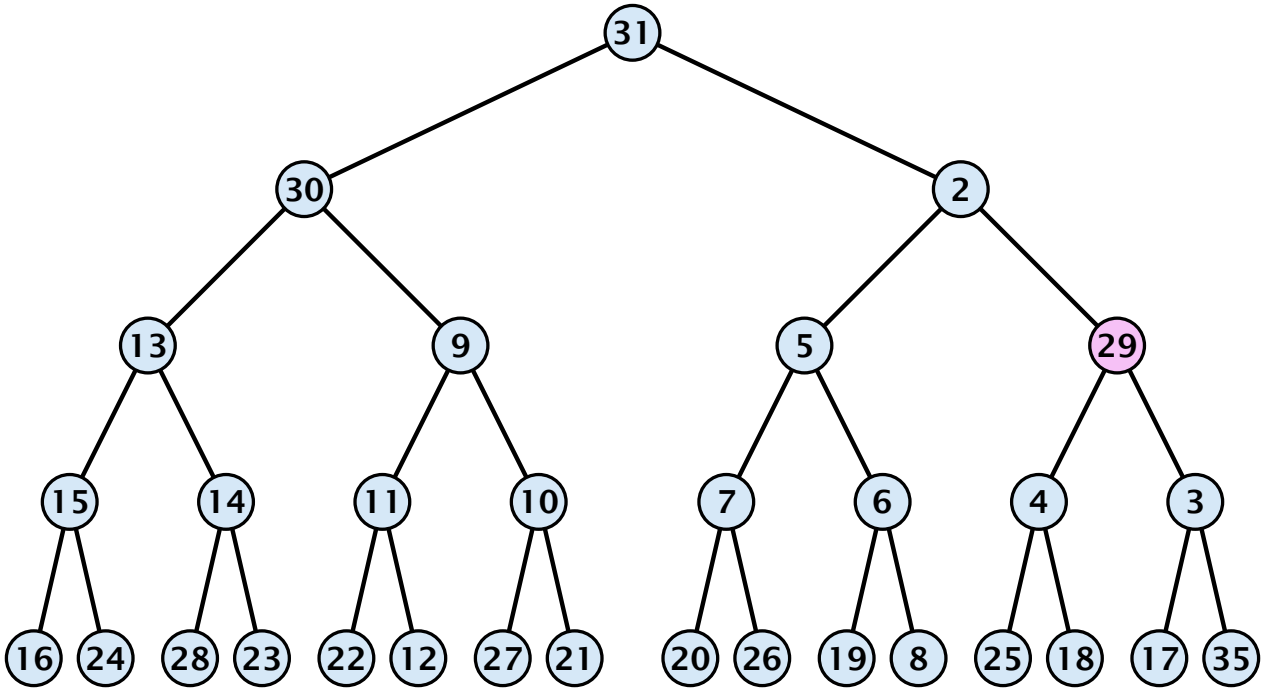
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

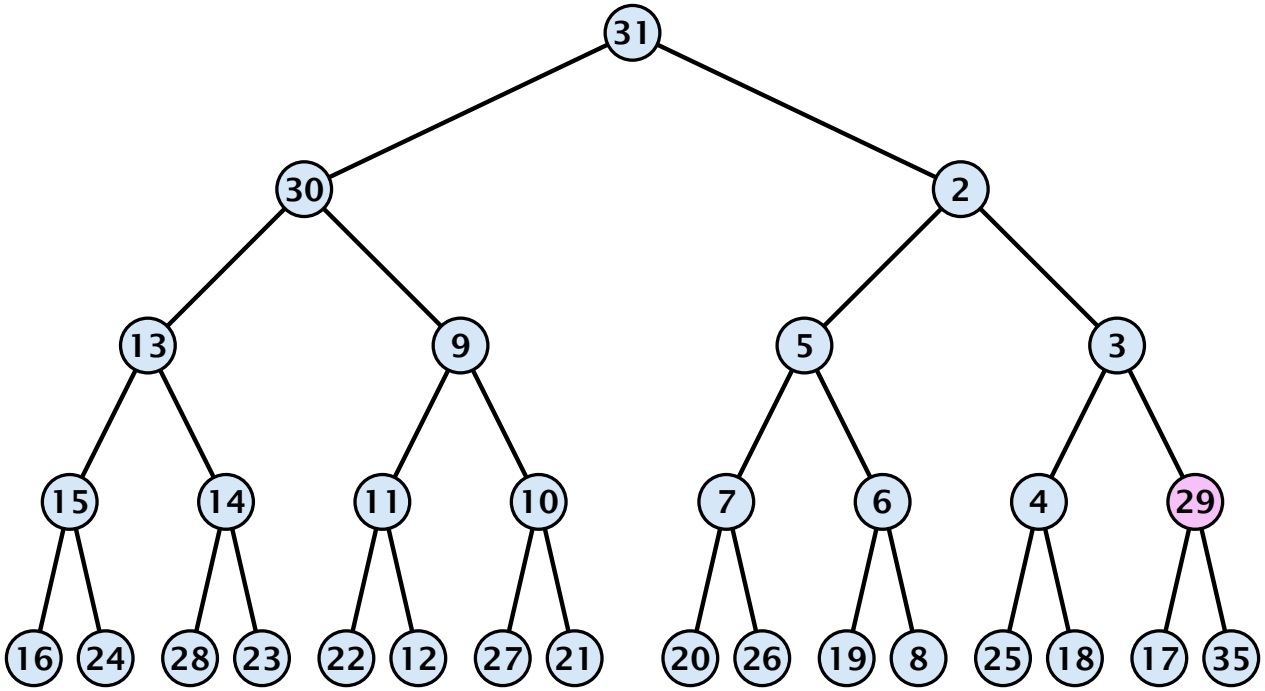
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

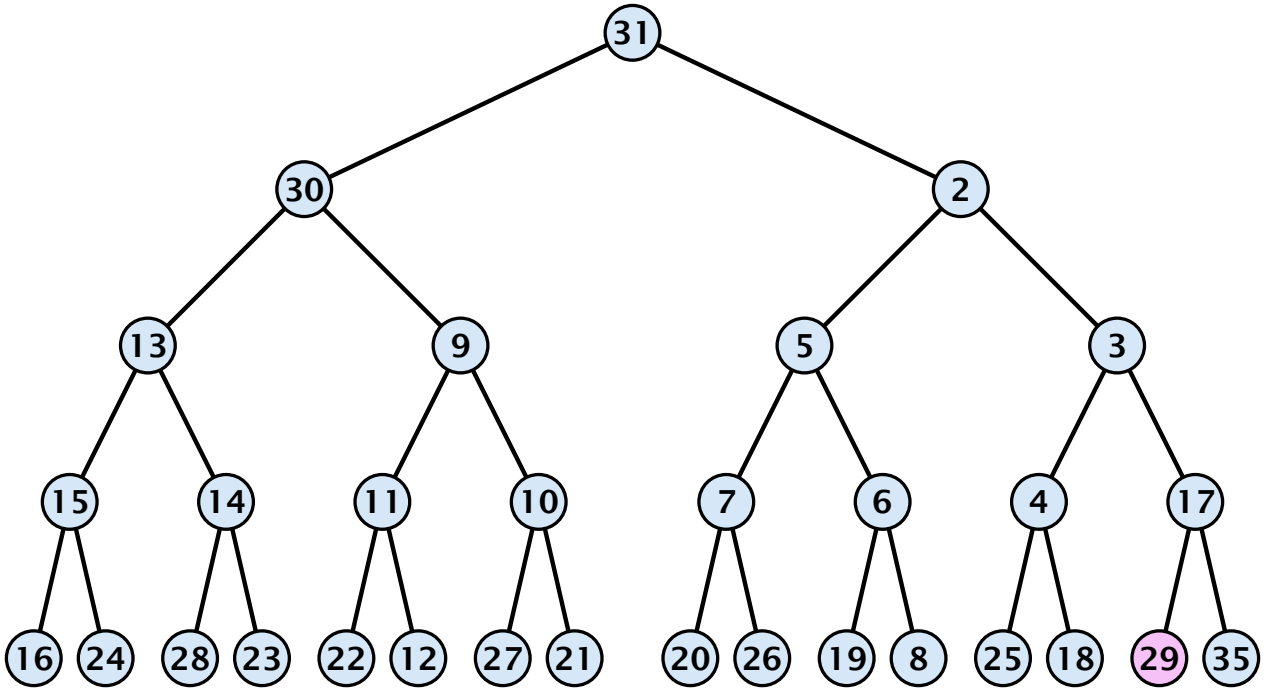
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

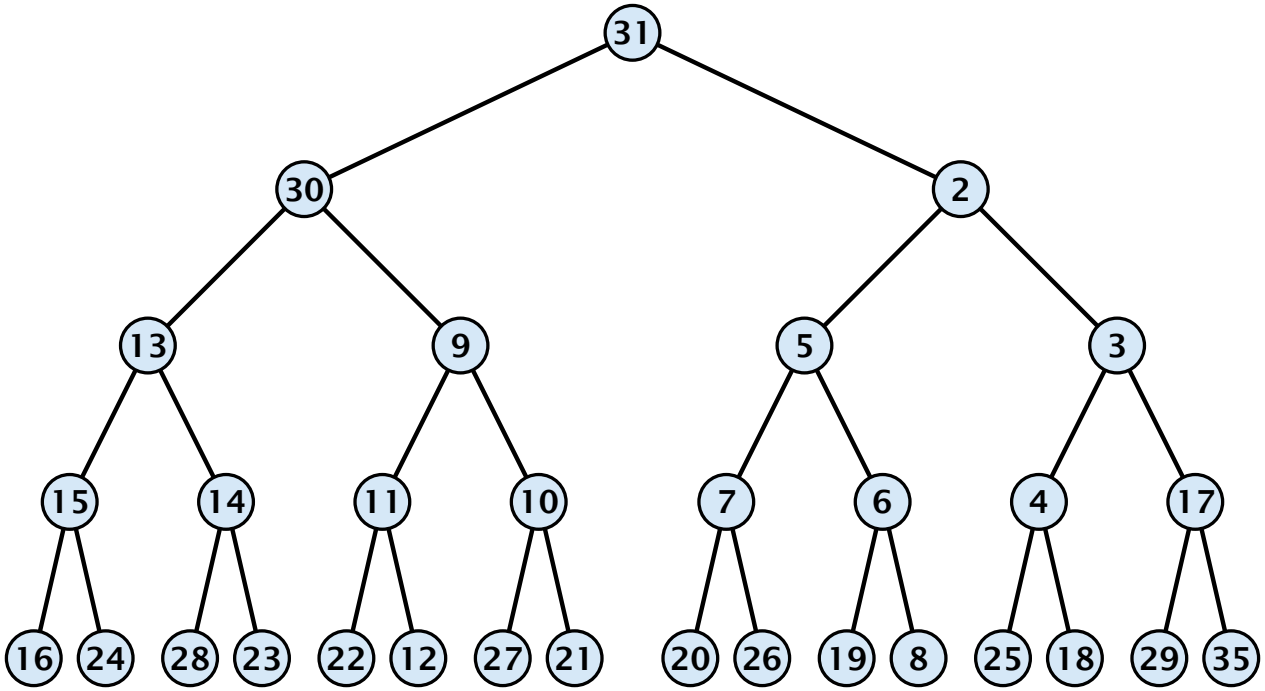
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

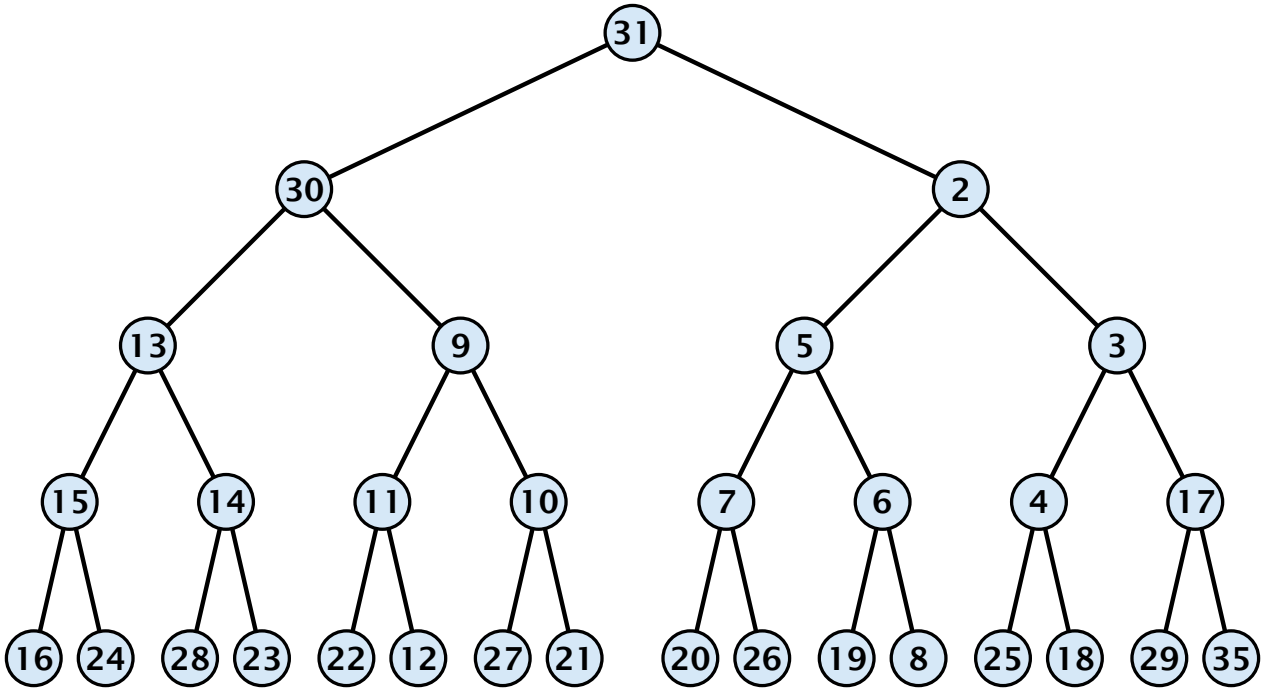
Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

Wir können einen Heap in Linearzeit erzeugen:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \sum_{i} i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Die Analyse auf der vorherigen Folie betrachtet nur die Kosten für die sift-down Operationen.

1. Wie bekommt man alle Schlüssel in eine Baumstruktur?
2. Wie realisiert man die Reihenfolge der sift-down Operationen?

2. kann mit Hilfe einer BFS-Suche realisiert werden; man startet eine BFS und verkettet die Baumknoten gemäß der Reihenfolge.

Auch **1.** kann man durch eine BFS-artige Generierung des Baumes erreichen.

```
1 insertCompleteBinary(A, n)
2     Queue q;
3     q.enqueue(&root);
4     for (i=0; i<n; i++)
5         TreeNode*  n = new TreeNode(A[i]);
6         TreeNode** t = q.dequeue();
7         n->parent = *t;
8         q.enqueue(&(n->left));
9         q.enqueue(&(n->right));
```

Binäre Heaps

Operationen:

- **minimum():** Gib Wurzelelement zurück. Zeit $\mathcal{O}(1)$.
- **isEmpty():** Überprüfe ob Wurzelzeiger NULL. Zeit $\mathcal{O}(1)$.
- **insert(k):** füge bei Nachfolger von x ein. bubble up. Zeit $\mathcal{O}(\log n)$.
- **delete(h):** tausche mit x; bubble up **or** sift-down. Zeit $\mathcal{O}(\log n)$.
- **build($x_1, \ldots, x_n$):** Füge Elemente beliebig ein; führe sift-down-Operationen durch; starte mit den untersten Leveln. Zeit $\mathcal{O}(n)$.

Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des i-ten Elementes findet man an Position $[\frac{i-1}{2}]$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von x zu finden ist viel einfacher als auf den vorherigen Folien. x wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des i-ten Elementes findet man an Position $\lfloor \frac{i-1}{2} \rfloor$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von x zu finden ist viel einfacher als auf den vorherigen Folien. x wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des i-ten Elementes findet man an Position $\lfloor \frac{i-1}{2} \rfloor$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von x zu finden ist viel einfacher als auf den vorherigen Folien. x wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

Binäre Heaps

Die Standardimplementierung eines Binärheaps speichert den Binärbaum in einem Array Sei $A[0,\ldots,n-1]$ das Array

- Das Elternelement des i-ten Elementes findet man an Position $\lfloor \frac{i-1}{2} \rfloor$.
- Das linke Kind findet man an Position $2i+1$.
- Das rechte Kind an $2i+2$.

Den Nachfolger von x zu finden ist viel einfacher als auf den vorherigen Folien. x wird einfach um eins erhöht.

Die resultierende Warteschlange ist nicht adressierbar. Die Elemente behalten ihre Position nicht und deshalb gibt es keine stabilen Handles.

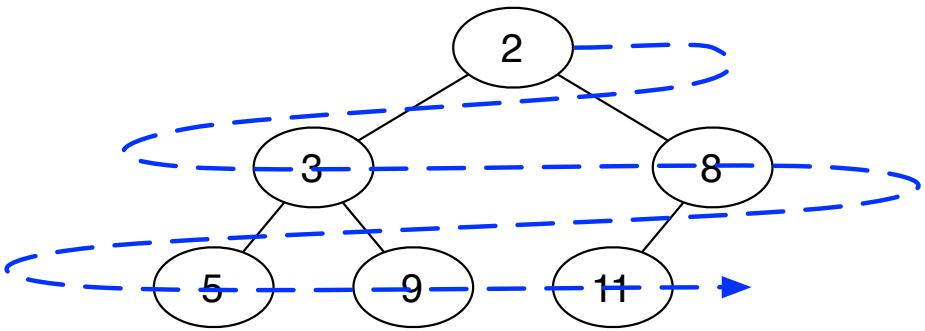
Binärbaum als Array

- vollständiger Binärbaum Höhe k hat $2^{k+1}-1$ Knoten

  $\Rightarrow$ speichere Knoten von oben nach unten, von links nach rechts in Array

  $\Rightarrow$ maximale Größe des Arrays: $2^{k+1}-1$

- Beispiel fast vollständiger Binärbaum:



Binärbaum als Array

Wurzel an Position 0.

Knoten an Position i:

- Elternknoten an Position $\lfloor (i-1)/2 \rfloor$
- linkes Kind an Position $2i+1$;
- rechtes Kind an Position $2i+2$