

16. Natürliche Suchbäume

[Ottman/Widmayer, Kap. 5.1, Cormen et al, Kap. 12.1 - 12.3]

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit. **Manche Operationen gar nicht unterstützt:**

- Aufzählen von Schlüssel in aufsteigender Anordnung
- Nächst kleinerer Schlüssel zu gegebenem Schlüssel
- Schlüssel k in vorgegebenem Intervall $k \in [l, r]$

Bäume

Bäume sind

- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

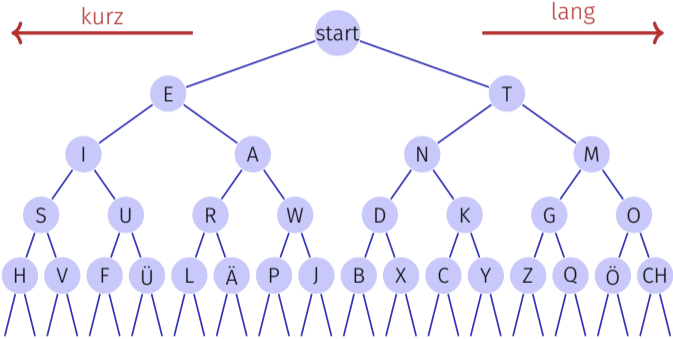
Bäume

Verwendung

- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffmann Code
- Suchbäume: Ermöglichen effizientes Suchen eines Elementes

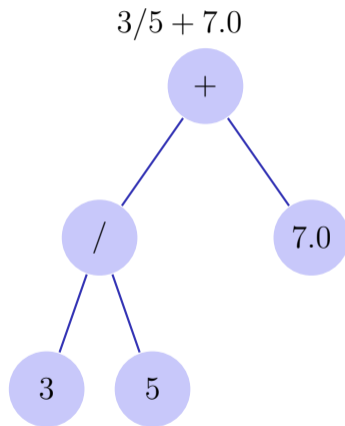


Beispiele



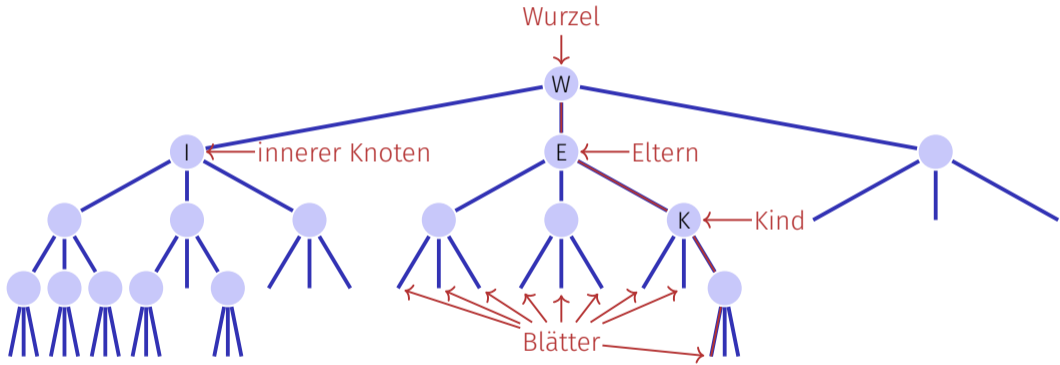
Morsealphabet

Beispiele



Ausdrucksbaum

Nomenklatur



- **Ordnung** des Baumes: Maximale Anzahl Kindknoten (hier: 3)
- **Höhe** des Baumes: Maximale Pfadlänge Wurzel zu Blatt (hier: 4)

Binäre Bäume

Ein *binärer Baum* ist

- entweder ein Blatt, d.h. ein leerer Baum,
- oder ein innerer Knoten mit zwei Bäumen T_l (linker Teilbaum) und T_r (rechter Teilbaum) als linken und rechten Nachfolger.

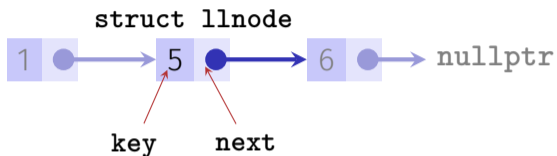
In jedem inneren Knoten v wird gespeichert

- ein Schlüssel $v.\mathbf{key}$ und
- zwei Zeiger $v.\mathbf{left}$ und $v.\mathbf{right}$ auf die Wurzeln der linken und rechten Teilbäume.



Ein Blatt wird durch den **null**-Zeiger repräsentiert

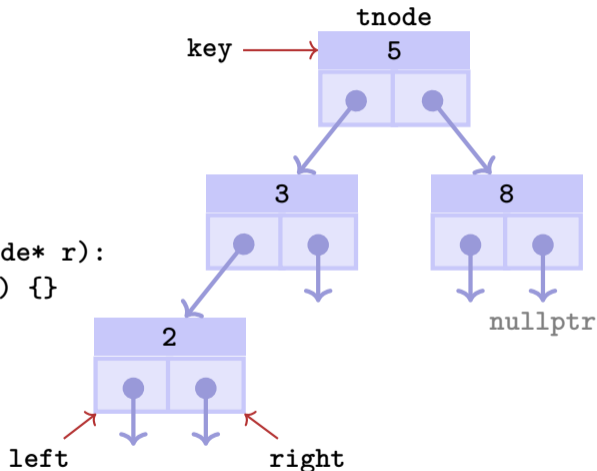
Zur Erinnerung: Listknoten in C++



```
struct llnode {  
    int key;  
    llnode* next;  
    llnode(int k, llnode* n): key(k), next(n) {} // Constructor  
};
```

Zur Erinnerung: Baumknoten in C++

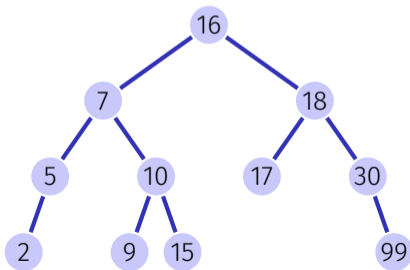
```
struct tnode {  
    int key;  
    tnode* left;  
    tnode* right;  
    tnode(int k, tnode* l, tnode* r):  
        key(k), left(l), right(r) {}  
};
```



Binärer Suchbaum

Ein *binärer Suchbaum* ist ein binärer Baum, der die **Suchbaumeigenschaft** erfüllt:

- Jeder Knoten v speichert einen Schlüssel
- Schlüssel im linken Teilbaum $v.\mathbf{left}$ kleiner als $v.\mathbf{key}$
- Schlüssel im rechten Teilbaum $v.\mathbf{right}$ grösser als $v.\mathbf{key}$



Suchen

Input: Binärer Suchbaum mit Wurzel r ,
Schlüssel k

Output: Knoten v mit $v.key = k$ oder **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.key$ **then**

 | **return** v

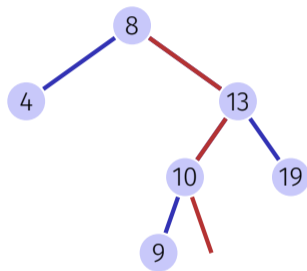
else if $k < v.key$ **then**

 | $v \leftarrow v.left$

else

 | $v \leftarrow v.right$

return null



Search (12) \rightarrow **null**

Suchen in C++

```
bool contains(const llnode* root, int search_key) {  
    while (root != nullptr) {  
        if (search_key == root->key) return true;  
        else if (search_key < root->key) root = root->left;  
        else root = root->right;  
    }  
  
    return false;  
}
```

Anmerkungen (pot. auch für späteren Code):

- **contains** wäre typischerweise eine Memberfunktion von **struct tnode** oder **class bin_search_tree** (→ leicht andere Signatur)
- Rekursive Implementation ebenfalls möglich

Höhe eines Baumes

Die Höhe $h(T)$ eines binären Baumes T mit Wurzel r ist gegeben als

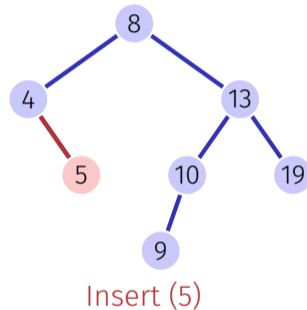
$$h(r) = \begin{cases} 0 & \text{falls } r = \mathbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist somit (im schlechtesten Fall) $\mathcal{O}(h(T))$

Einfügen eines Schlüssels

Einfügen des Schlüssels k

- Suche nach k
- Falls erfolgreich: z.B. Fehlerausgabe
- Falls erfolglos: Einfügen des Schlüssels am erreichten Blatt

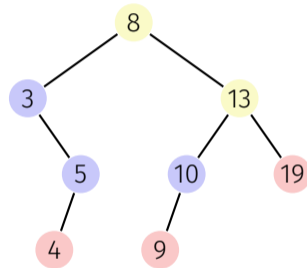


Knoten entfernen

Drei Fälle möglich

- Knoten hat keine Kinder
- Knoten hat ein Kind
- Knoten hat zwei Kinder

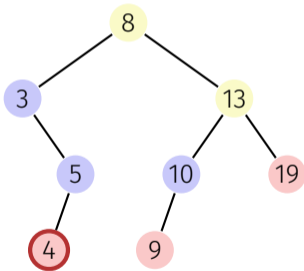
[Blätter zählen hier nicht]



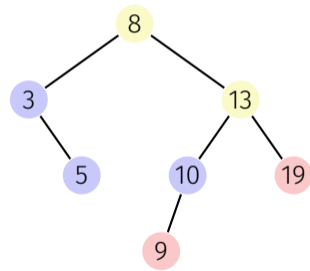
Knoten entfernen

Knoten hat keine Kinder

Einfacher Fall: Knoten durch Blatt ersetzen.



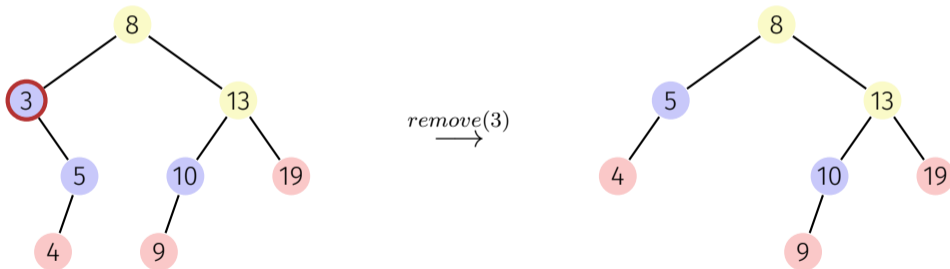
remove(4)
→



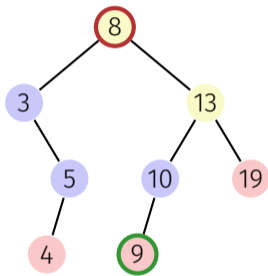
Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



Knoten entfernen



Knoten v hat zwei Kinder

Anforderungen an den Ersatzknoten w :

1. $w.key$ ist grösser als alle in Schlüssel in $v.left$
2. $w.key$ ist kleiner als alle in Schlüssel in $v.right$
3. hat idealerweise keine Kinder

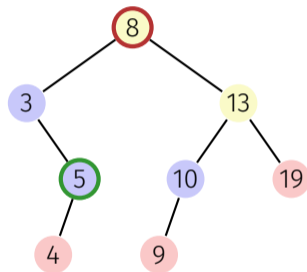
Beobachtung: Der kleinste Schlüssel im rechten Teilbaum $v.right$ (hier: 9) erfüllt Bedingungen 1, 2; und hat maximal ein (rechtes) Kind.

Lösung: Ersetze v durch ebenjenen *symmetrischen Nachfolger*.

Aus Symmetriegründen ...

Knoten v hat zwei Kinder

Auch möglich: ersetze v durch seinen *symmetrischen Vorgänger*



Algorithmus SymmetricSuccessor(v)

Input: Knoten v eines binären Suchbaumes

Output: Symmetrischer Nachfolger von v

$w \leftarrow v.\text{right}$

$x \leftarrow w.\text{left}$

while $x \neq \text{null}$ **do**

$w \leftarrow x$

$x \leftarrow x.\text{left}$

return w

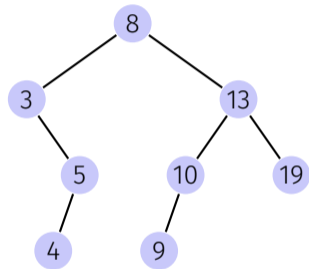
Analyse

Löschen eines Elementes v aus einem Baum T benötigt $\mathcal{O}(h(T))$
Elementarschritte:

- Suchen von v hat Kosten $\mathcal{O}(h(T))$
- Hat v maximal ein Kind ungleich **null**, dann benötigt das Entfernen $\mathcal{O}(1)$
- Das Suchen des symmetrischen Nachfolgers n benötigt $\mathcal{O}(h(T))$ Schritte.
Entfernen und Einfügen von n hat Kosten $\mathcal{O}(1)$

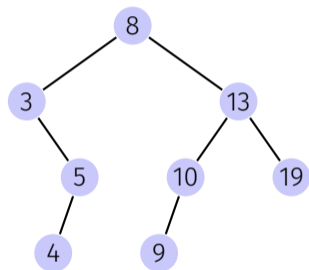
Traversierungsarten

- *Hauptreihenfolge (preorder)*:
 v , dann $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$.
8, 3, 5, 4, 13, 10, 9, 19
- *Nebenreihenfolge (postorder)*:
 $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$, dann v .
4, 5, 3, 9, 10, 19, 13, 8
- *Symmetrische Reihenfolge (inorder)*:
 $T_{\text{left}}(v)$, dann v , dann $T_{\text{right}}(v)$.
3, 4, 5, 8, 9, 10, 13, 19

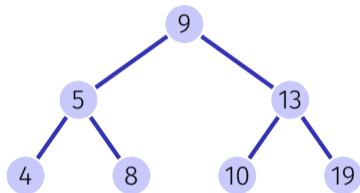


Weitere unterstützte Operationen

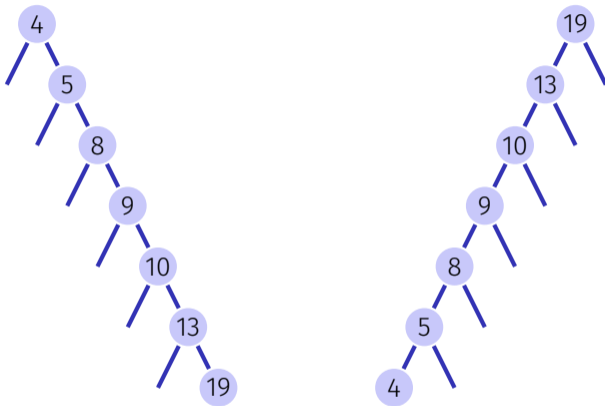
- $Min/Max(T)$: Auslesen des Minimums/Maximums in $\mathcal{O}(h(T))$
- $ExtractMin/Max(T)$: Auslesen und Entfernen des Min/Max in $\mathcal{O}(h(T))$
- $List(T)$: Ausgeben einer sortierten Liste der Elemente von T
- $Join(T_1, T_2)$: Zusammenfügen zweier Bäume mit $Max(T_1) < Min(T_2)$ in $\mathcal{O}(h(T_1, T_2))$



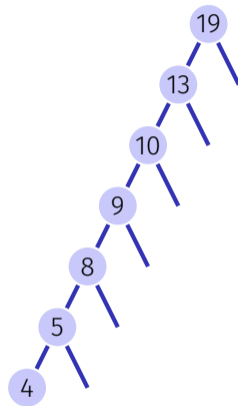
Suchbäume: balanciert vs. degeneriert



insert 9,5,13,4,8,10,19:
bestmöglich
balanciert



insert 4,5,8,9,10,13,19:
lineare Liste



insert 19,13,10,9,8,5,4:
lineare Liste

Probabilistisch

Ein Suchbaum, welcher aus einer zufälligen Sequenz von Zahlen erstellt wird, hat erwartete Pfadlänge von $\mathcal{O}(\log n)$.

Achtung: das gilt nur für Einfügeoperation. Wird der Baum zufällig durch Einfügen und Entfernen gebildet, ist die erwartete Pfadlänge $\mathcal{O}(\sqrt{n})$.

Balancierte Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine $\mathcal{O}(\log n)$ Worst-Case-Garantie.

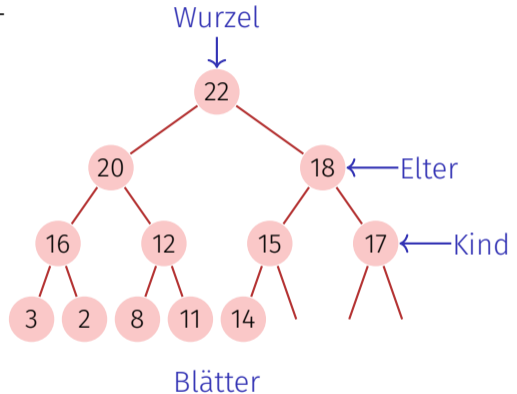
17. Heaps

Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap*

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. **Heap-Bedingung:**
Max-(Min-)Heap: Schlüssel eines Kindes kleiner (größer) als der des Elternknotens



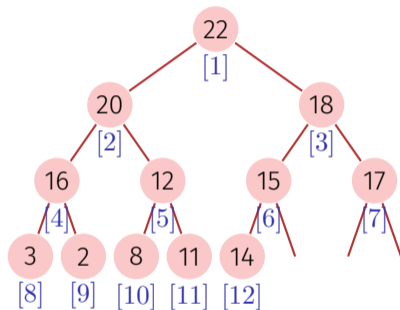
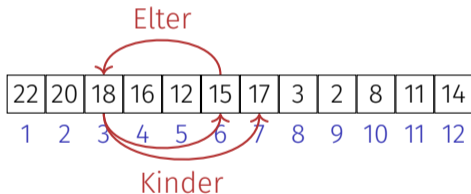
*Heap (Datenstruktur), nicht wie in „Heap und Stack“ (Speicherallokation)

Heap als Array

Baum \rightarrow Array:

■ $\text{Kinder}(i) = \{2i, 2i + 1\}$

■ $\text{Elter}(i) = \lfloor i/2 \rfloor$



Abhängig vom Startindex!²¹

²¹Für Arrays, die bei 0 beginnen: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

Höhe eines Heaps

Welche Höhe $H(n)$ hat ein Heap mit n Knoten? Auf der i -ten Ebene eines Binärbaumes befinden sich höchstens 2^i Knoten. Bis auf die letzte Ebene sind alle Ebenen eines Heaps gefüllt.

$$H(n) = \min\{h \in \mathbb{N} : \sum_{i=0}^{h-1} 2^i \geq n\}$$

Mit $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

also

$$H(n) = \lceil \log_2(n + 1) \rceil.$$

Heap in C++

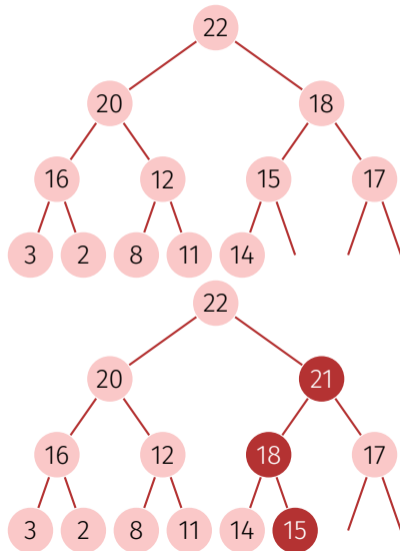
```
class MaxHeap {
    int* keys; // Pointer to first key
    unsigned int capacity; // Length of key array
    unsigned int count; // Keys in use <= capacity
    // Or even better: build on top of std::vector

public:
    MaxHeap(unsigned int initial_capacity):
        keys(new int[initial_capacity]),
        capacity(initial_capacity),
        count(0)
    {}

    void insert(unsigned int key) { ...}
    int remove_max() { ...}
    ...
}
```

Einfügen

- Füge neuen Schlüssel an erster freien Stelle ein. Verletzt Heap-Eigenschaft potenziell.
- Stelle Heap-Eigenschaft wieder her: Sukzessives Aufsteigen
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Algorithmus Aufsteigen(A, m)

Input: Array A mit mindestens m Schlüsseln und Heapstruktur auf $A[1, \dots, m-1]$

Output: Array A mit Heapstruktur auf $A[1, \dots, m]$

$v \leftarrow A[m]$ // Neuer Schlüssel

$c \leftarrow m$ // Index derzeitiger Knoten (child)

$p \leftarrow \lfloor c/2 \rfloor$ // Index Elternknoten (parent)

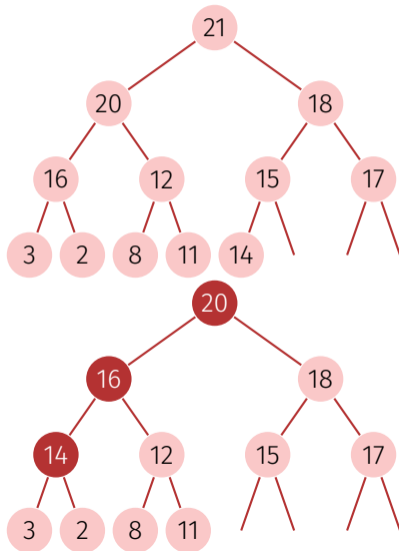
while $c > 1$ and $v > A[p]$ **do**

$A[c] \leftarrow A[p]$ // Schlüssel Elternknoten \rightarrow Schlüssel derzeitiger Knoten
 $c \leftarrow p$ // Elternknoten \rightarrow derzeitiger Knoten
 $p \leftarrow \lfloor c/2 \rfloor$

$A[c] \leftarrow v$ // Neuen Schlüssel platzieren

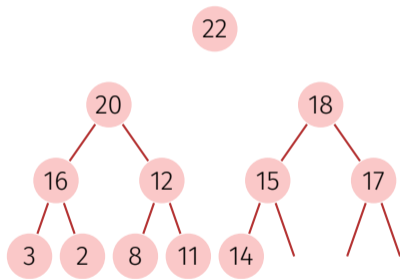
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



Algorithmus Versickern(A, i, m)

Input: Array A mit Heapstruktur für die Kinder von i . Letztes Element m .

Output: Array A mit Heapstruktur für i mit letztem Element m .

while $2i \leq m$ **do**

$j \leftarrow 2i$; // j linkes Kind

if $j < m$ and $A[j] < A[j + 1]$ **then**

$j \leftarrow j + 1$; // j rechtes Kind mit grösserem Schlüssel

if $A[i] < A[j]$ **then**

 Vertausche($A[i], A[j]$)

$i \leftarrow j$; // weiter versickern

else

$i \leftarrow m$; // versickern beendet

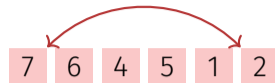
Heaps Sortieren

Sei $A[1, \dots, n]$ ein Heap.

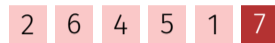
Solange $n > 1$:

1. $Vertausche(A[1], A[n])$
2. $Versickere(A, 1, n - 1)$
3. $n \leftarrow n - 1$

Tauschen \Rightarrow



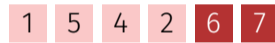
Versickern \Rightarrow



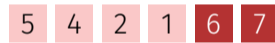
Tauschen \Rightarrow



Versickern \Rightarrow



Tauschen \Rightarrow



Versickern \Rightarrow



Tauschen \Rightarrow



Versickern \Rightarrow



Tauschen \Rightarrow



Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

Algorithmus HeapSort(A, n)

Input: Array A der Länge n .

Output: A sortiert.

// Heap aufbauen

for $i \leftarrow n/2$ **downto** 1 **do**

└ Versickere(A, i, n)

// Nun ist A ein Heap

for $i \leftarrow n$ **downto** 2 **do**

└ Vertausche($A[1], A[i]$)

└ Versickere($A, 1, i - 1$)

// Nun ist A sortiert.

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. \Rightarrow Sortieren eines Heaps kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen beim Sortieren eines Heaps auch $\mathcal{O}(n \log n)$.

Analyse: Heap aufbauen

Aufrufe von *Versickere*: $n/2$.

Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade sind aber im Mittel viel kürzer:

Wir verwenden, dass $h(n) = \lceil \log_2 n + 1 \rceil = \lfloor \log_2 n \rfloor + 1$ für $n > 0$

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level } l} \cdot \underbrace{(\lfloor \log_2 n \rfloor + 1 - l - 1)}_{\text{Höhe Heaps auf Level } l} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k \\ &= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n) \end{aligned}$$

mit $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ($0 < x < 1$) und $s(\frac{1}{2}) = 2$

Nachteile

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

- ⚠️ Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache-Effekt).
- ⚠️ Zwei Vergleiche vor jeder benötigten Bewegung.