18. 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:

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

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

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

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

Bäume

Bäume sind

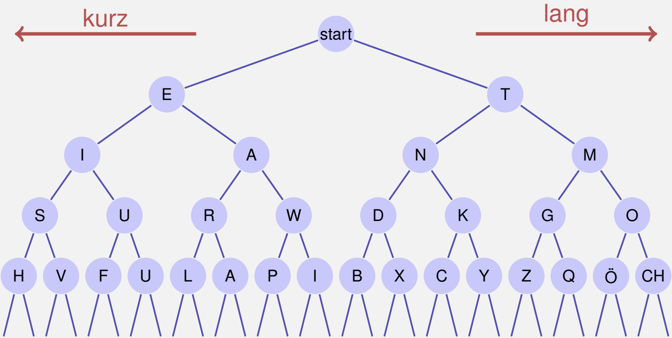
- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

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

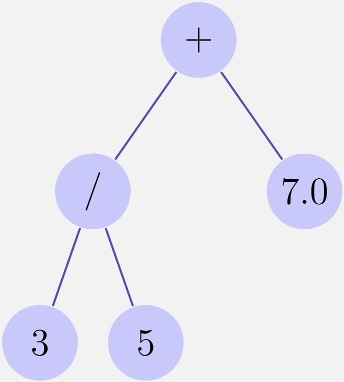
Beispiele



Morsealphabet

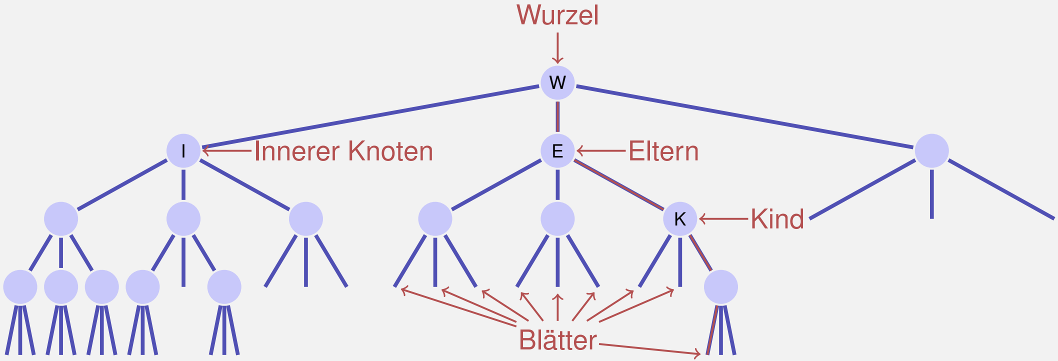
Beispiele

$$3/5 + 7.0$$



Ausdrucksbaum

Nomenklatur



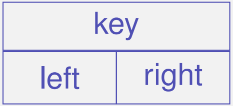
- Ordnung des Baumes: Maximale Anzahl Kindknoten, hier: 3
- Höhe des Baumes: maximale Pfadlänge Wurzel – Blatt (hier: 4)

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 Knoten v speichern wir



- einen Schlüssel $v.\mathrm{key}$ und
- zwei Zeiger $v.\mathrm{left}$ und $v.\mathrm{right}$ auf die Wurzeln der linken und rechten Teilbäume.
- Ein Blatt wird durch den **null**-Zeiger repräsentiert

Baumknoten in Java

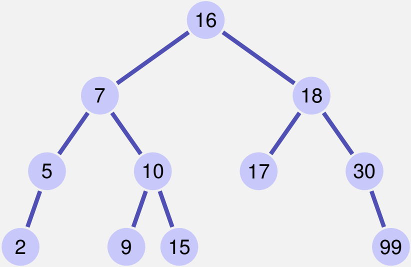
```java
public class SearchNode {
  int key;      // Schluessel
  SearchNode left;    // linker Teilbaum
  SearchNode right;   // rechter Teilbaum

  // Konstruktor: Knoten ohne Nachfolger
  SearchNode(int k){
    key = k;
    left = right = null;
  }
}
```

Binärer Suchbaum

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

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



Suchen

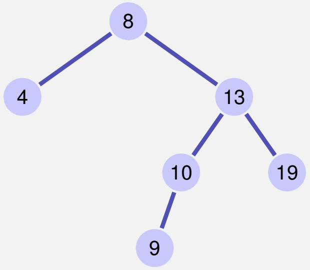
Input : Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output : Knoten v mit $v.\text{key} = k$ oder **null**

$v \leftarrow r$

while $v \neq$ **null do**
    if $k = v.\text{key}$ **then**
        return v
    else if $k < v.\text{key}$ **then**
        $v \leftarrow v.\text{left}$
    else
        $v \leftarrow v.\text{right}$

return null



Suchen

Input : Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output : Knoten v mit $v.\mathrm{key} = k$ oder **null**

$v \leftarrow r$

while $v \neq$ **null do**

  if $k = v.\mathrm{key}$ **then**

    return v

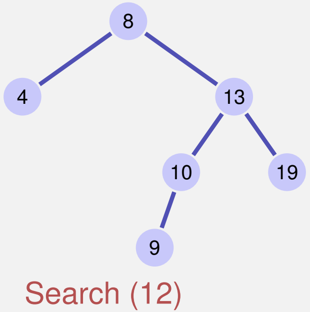
  else if $k < v.\mathrm{key}$ **then**

    $v \leftarrow v.\mathrm{left}$

  else

    $v \leftarrow v.\mathrm{right}$

return null



Suchen

Input : Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output : Knoten v mit $v.\mathrm{key} = k$ oder **null**

$v \leftarrow r$

while $v \neq$ **null do**

  if $k = v.\mathrm{key}$ **then**

    return v

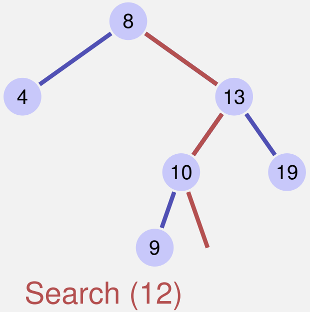
  else if $k < v.\mathrm{key}$ **then**

    $v \leftarrow v.\mathrm{left}$

  else

    $v \leftarrow v.\mathrm{right}$

return null



Suchen

Input : Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output : Knoten v mit $v.\text{key} = k$ oder **null**

$v \leftarrow r$

while $v \neq$ **null do**

    if $k = v.\text{key}$ **then**

        return v

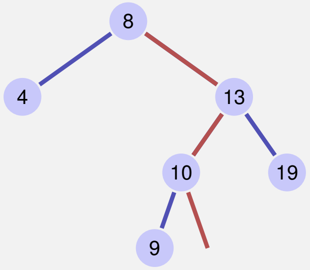
    else if $k < v.\text{key}$ **then**

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

    else

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

return null



Search (12) $\rightarrow$ **null**

Suchbaum und Suchen in Java

```java
public class SearchTree {
  SearchNode root = null; // Wurzelknoten

  // Gibt Knoten mit Schluessel k zurueck.
  // Wenn nicht existiert: null.
  public SearchNode Search (int k){
    SearchNode n = root;
    while (n != null && n.key != k){
      if (k < n.key) n = n.left;
      else n = n.right;
    }
    return n;
  }
  ... // Einfuegen, Loeschen
}
```

Höhe eines Baumes

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

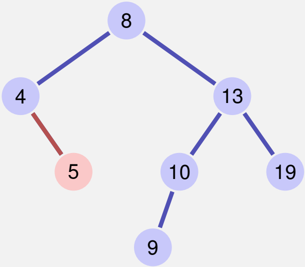
$$h(r) = \begin{cases} 0 & \text{falls } r = \textbf{null} \\ 1 + \max\{h(r.\mathrm{left}), h(r.\mathrm{right})\} & \text{sonst.} \end{cases}$$

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

Einfügen eines Schlüssels

Einfügen des Schlüssels k

- Suche nach k.
- Wenn erfolgreich: Fehlerausgabe
- Wenn erfolglos: Einfügen des Schlüssels am erreichten Blatt.
- Implementation: der Teufel steckt im Detail



Insert (5)

Knoten Einfügen in Java

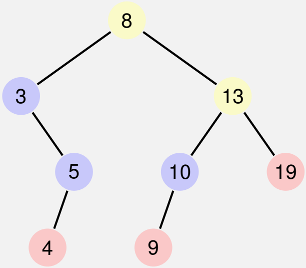
```
public SearchNode Insert (int k) {
  if (root == null) { return root = new SearchNode(k); }
  SearchNode t=root;
  while (true) {
    if (k == t.key) { return null; }
    if (k < t.key) {
      if (t.left == null) { return t.left = new SearchNode(k); }
      else { t = t.left; }
    }
    else { // k > t.key
      if (t.right == null) { return t.right = new SearchNode(k); }
      else { t = t.right; }
    }
  }
}
```

Knoten entfernen

Drei Fälle möglich

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

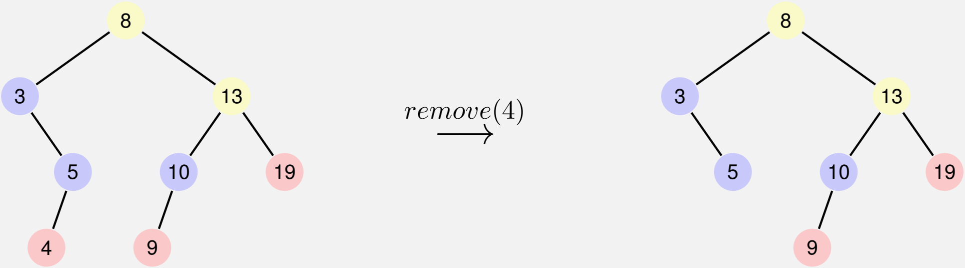
[Blätter zählen hier nicht]



Knoten entfernen

Knoten hat keine Kinder

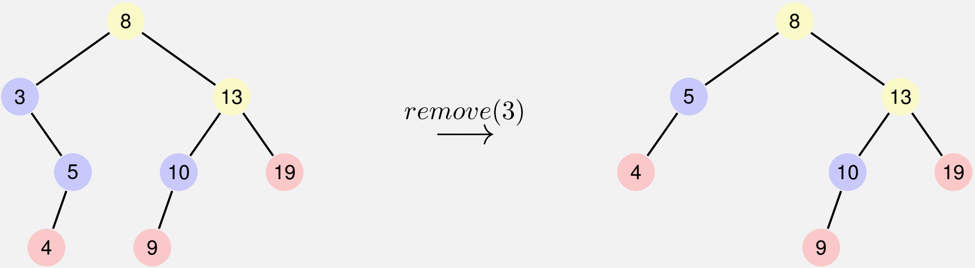
Einfacher Fall: Knoten durch Blatt ersetzen.



Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



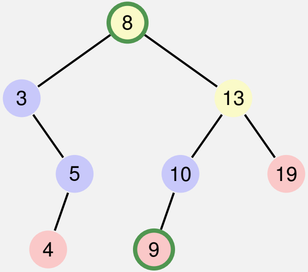
Knoten entfernen

Knoten v hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum $v.\mathrm{right}$ (der *symmetrische Nachfolger* von v)

- ist kleiner als alle Schlüssel in $v.\mathrm{right}$
- ist grösser als alle Schlüssel in $v.\mathrm{left}$
- und hat kein linkes Kind.

Lösung: ersetze v durch seinen symmetrischen Nachfolger

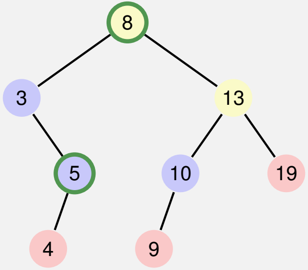


Aus Symmetriegründen...

Knoten v hat zwei Kinder

Auch möglich: ersetze v durch seinen symmetrischen Vorgänger

Implementation: der Teufel steckt im Detail!



Algorithmus SymmetricSuccessor(v)

Input : Knoten v eines binären Suchbaumes
Output : Symmetrischer Nachfolger von v
$w \leftarrow v.\mathrm{right}$
$x \leftarrow w.\mathrm{left}$
while $x \neq$ **null do**
    $w \leftarrow x$
    $x \leftarrow x.\mathrm{left}$

return w

SymmetricDesc in Java

```java
public SearchNode SymmetricDesc(SearchNode node) {
  if (node.left == null) { return node.right; }
  if (node.right == null) { return node.left; }
  SearchNode n = node;
  SearchNode parent = null;
  n = n.right;
  while (n.left != null) { parent = n; n = n.left; }
  if (parent != null) {
    parent.left = n.right;
    n.left = node.left;
    n.right = node.right;
  } else { n.left = node.left; }
  return n;
}
```

Dieser Algorithmus gibt den symmetrischen Nachfolger zurück. Aber tut noch mehr: er behandelt auch die Fälle mit einem oder keinem Nachfolger. Ausserdem entfernt er den Symmetrischen Nachfolger und setzt dessen Nachfolgeknoten.

Knoten Löschen in Java

```java
public void Delete (int k) {
  SearchNode n = root;
  if (n != null && n.key == k) {
    root = SymmetricDesc(root);
  } else {
    while (n != null) {
      if (n.left != null && k == n.left.key) {
        n.left = SymmetricDesc(n.left); return;
      } else if (n.right != null && k == n.right.key) {
        n.right = SymmetricDesc(n.right); return;
      } else if (k < n.key) { n = n.left;
      } else { n = n.right; }
    }
  }
}
```

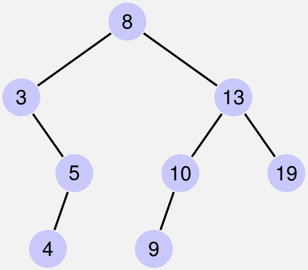
Analyse

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 $\mathcal{O}(1)$

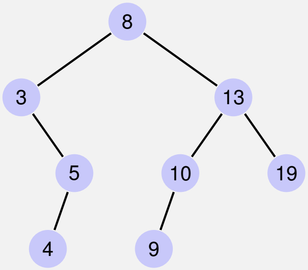
Traversierungsarten

- Hauptreihenfolge (preorder): v, dann $T_{\mathrm{left}}(v)$, dann $T_{\mathrm{right}}(v)$.



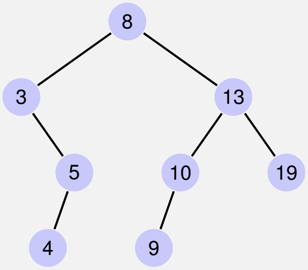
Traversierungsarten

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



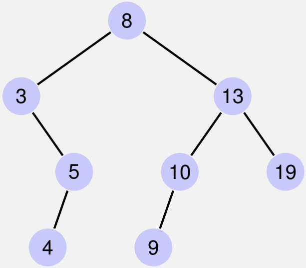
Traversierungsarten

- 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.



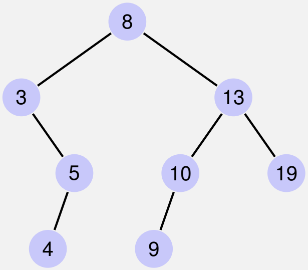
Traversierungsarten

- 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



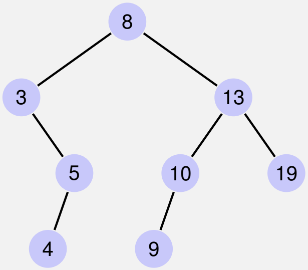
Traversierungsarten

- 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 $T_{\text{right}}(v)$.

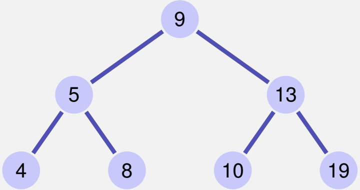


Traversierungsarten

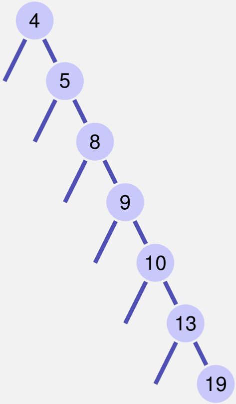
- Hauptreihenfolge (preorder): v, dann $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder): $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$, dann v.
  4, 5, 3, 9, 10, 19, 13, 8
- Symmetrische Reihenfolge (inorder): $T_{\text{left}}(v)$, dann v,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



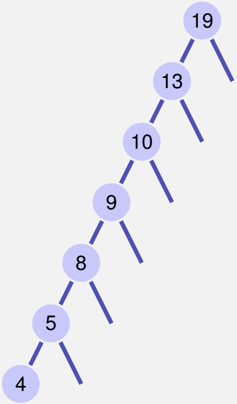
Degenerierte Suchbäume



Insert 9,5,13,4,8,10,19
bestmöglich
balanciert



Insert 4,5,8,9,10,13,19
Lineare Liste



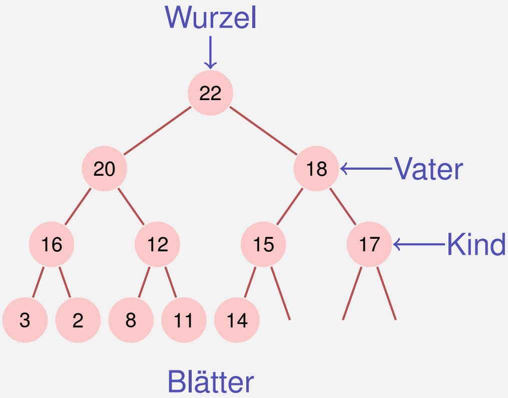
Insert 19,13,10,9,8,5,4
Lineare Liste

19. Heaps

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

[Max-]Heap[10]

Binärer Baum mit folgenden Eigenschaften

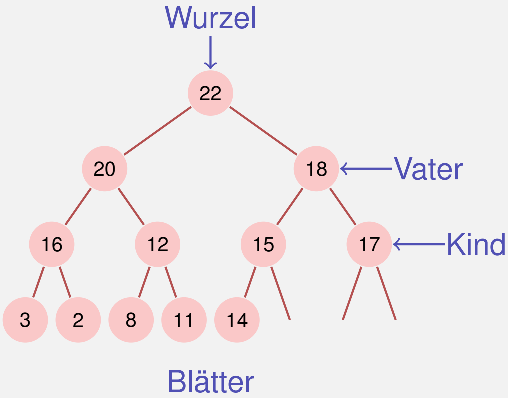


[10] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[10]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene

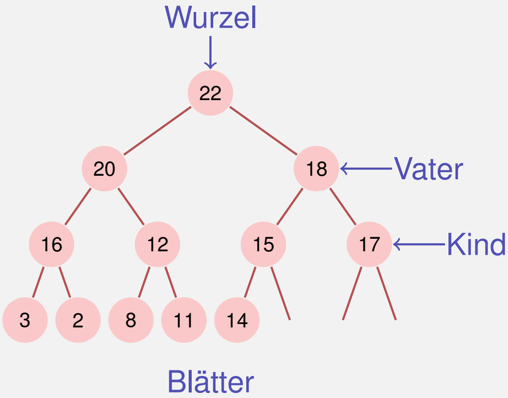


[10] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[10]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.

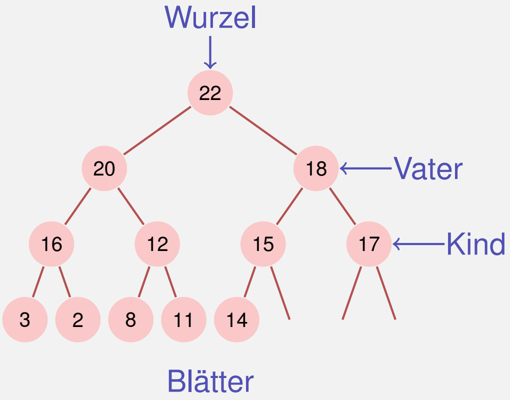


[10] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[10]

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 Vaters

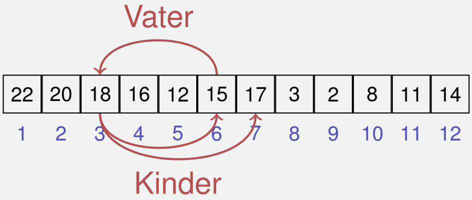


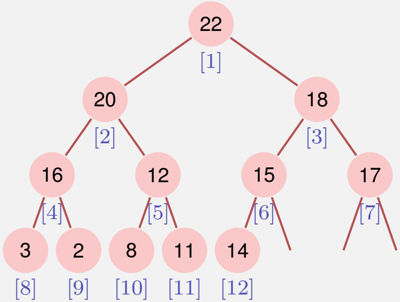
[10] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

Heap und Array

Baum $\rightarrow$ Array:

- Kinder$(i) = \{2i, 2i+1\}$
- Vater$(i) = \lfloor i/2 \rfloor$



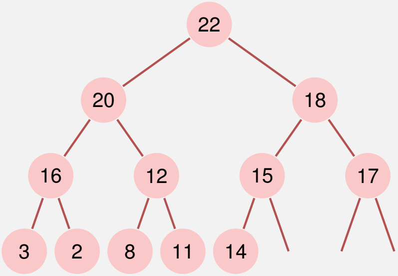


Abhängig von Startindex![11]

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

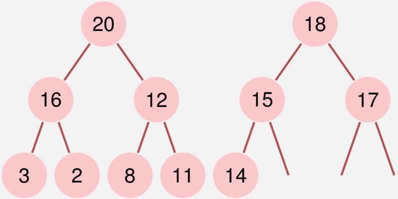
Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:

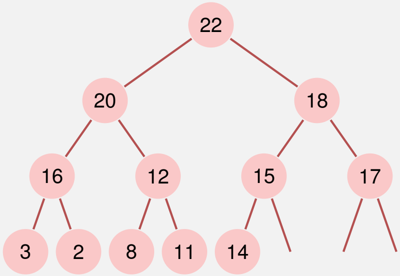


Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:

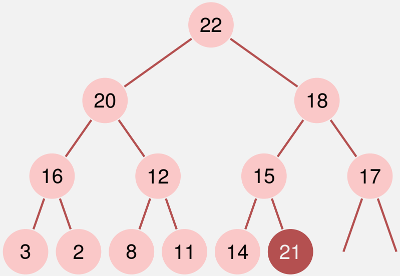


Einfügen



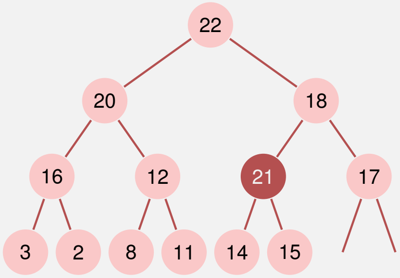
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.



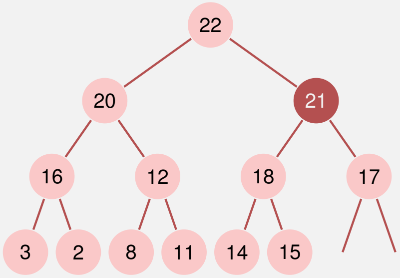
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



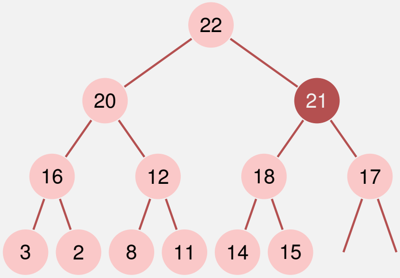
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.

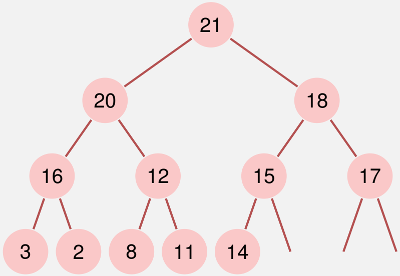


Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$

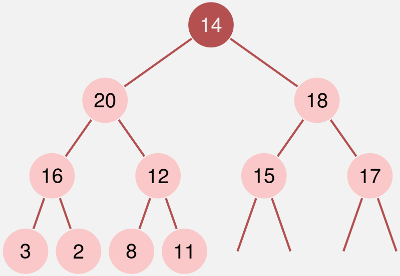


Maximum entfernen



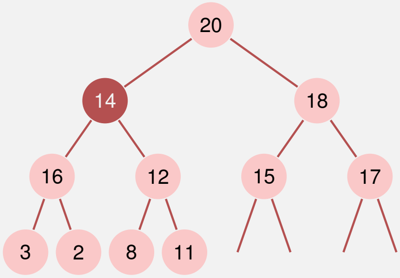
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



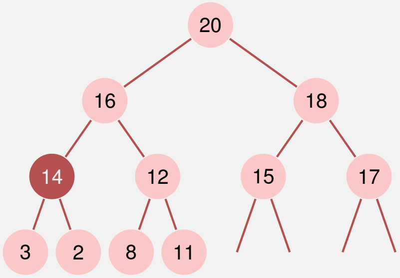
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



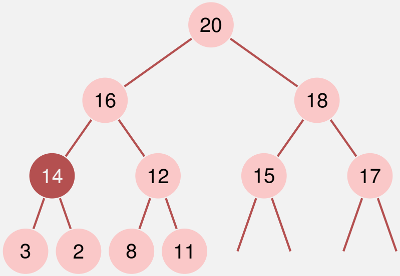
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



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: $\mathcal{O}(\log n)$



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 ≤ m do
    j ← 2i; // j linkes Kind
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j rechtes Kind mit grösserem Schlüssel
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // weiter versickern
    else
        i ← m; // versickern beendet
```

Heap Sortieren

7	6	4	5	1	2

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n - 1$);
- $n \leftarrow n - 1$

Heap Sortieren

	7	6	4	5	1	2
Tauschen $\Rightarrow$	2	6	4	5	1	7

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere$(A, 1, n - 1)$;
- $n \leftarrow n - 1$

Heap Sortieren

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

Heap Sortieren

$A[1,...,n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere$(A, 1, n-1)$;
- $n \leftarrow n - 1$

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7
Tauschen	$\Rightarrow$	1	5	4	2	6	7

Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7
Tauschen	$\Rightarrow$	1	5	4	2	6	7
Versickern	$\Rightarrow$	5	4	2	1	6	7
Tauschen	$\Rightarrow$	1	4	2	5	6	7
Versickern	$\Rightarrow$	4	1	2	5	6	7
Tauschen	$\Rightarrow$	2	1	4	5	6	7
Versickern	$\Rightarrow$	2	1	4	5	6	7
Tauschen	$\Rightarrow$	1	2	4	5	6	7

Heap erstellen

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

Folgerung:

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.

for $i \leftarrow n/2$ **downto** 1 **do**
    Versickere(A, i, n);
// Nun ist A ein Heap.
for $i \leftarrow n$ **downto** 2 **do**
    swap($A[1], A[i]$)
    Versickere($A, 1, i-1$)
// Nun ist A sortiert.

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. $\Rightarrow$ Heap sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade aber im Mittel viel kürzer, also sogar:

$$v(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot c \cdot h \in \mathcal{O}(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h})$$

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

$$v(n) \in \mathcal{O}(n).$$