Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München

Programm heute

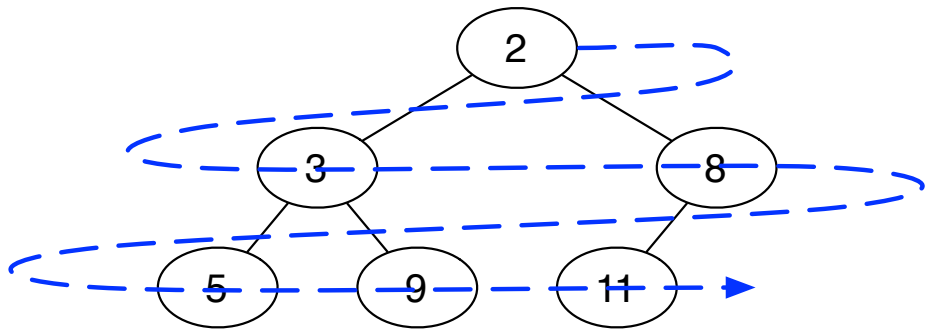
7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume

Binärbaum als sequentielle Liste I

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

  $\rightarrow$ speichere Knoten von oben nach unten, von links nach rechts in sequentieller Liste (Array)



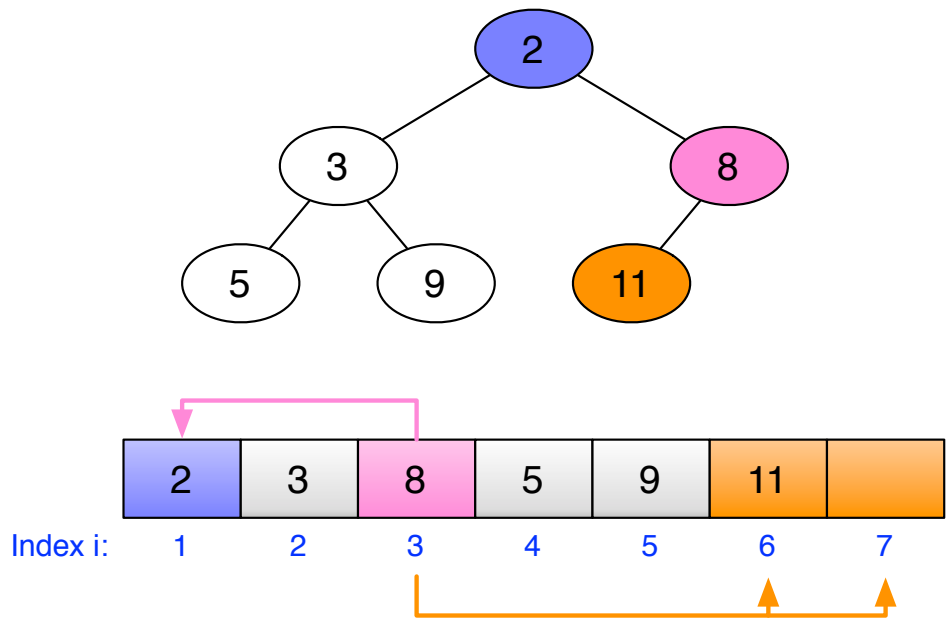
2	3	8	5	9	11	

Binärbaum als sequentielle Liste II

- Wurzel: an Position 1
- Knoten an Position i:
  - Vater-Knoten an Position $\lfloor i/2 \rfloor$
  - linkes Kind an Position $2i$;
  - rechtes Kind an Position $2i + 1$

Pseudocode:

- **vater**(i): **return** $\lfloor i/2 \rfloor$;
- **links**(i): **return** $2i$;
- **rechts**(i): **return** $2i + 1$;



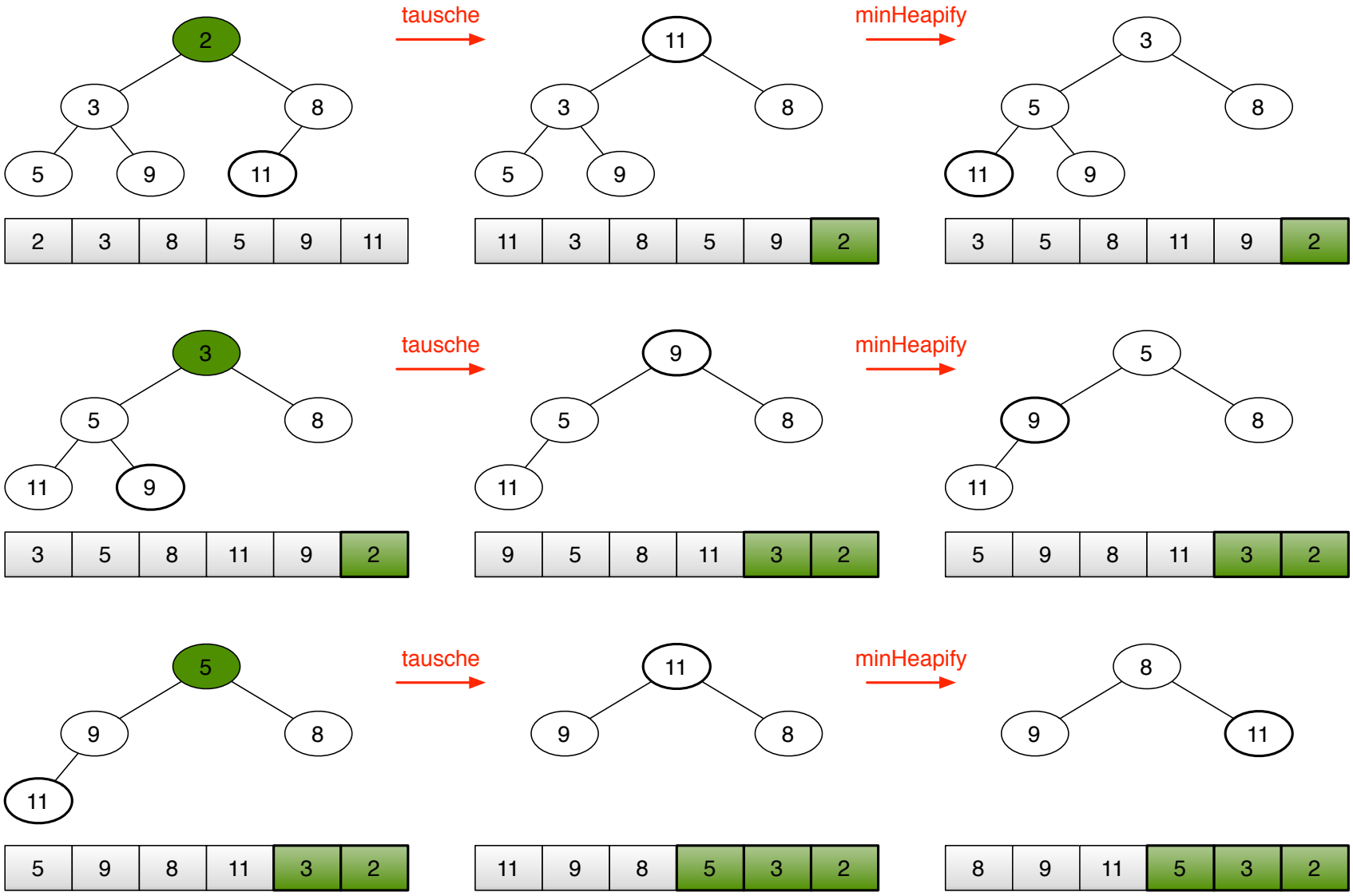
HeapSort

- Sortieren mit Heap
- Idee:
  - Heap erstellen mit **buildMinHeap**
  - wiederhole **extractMin** bis Heap leer
- mit Heap direkt im Eingabefeld:

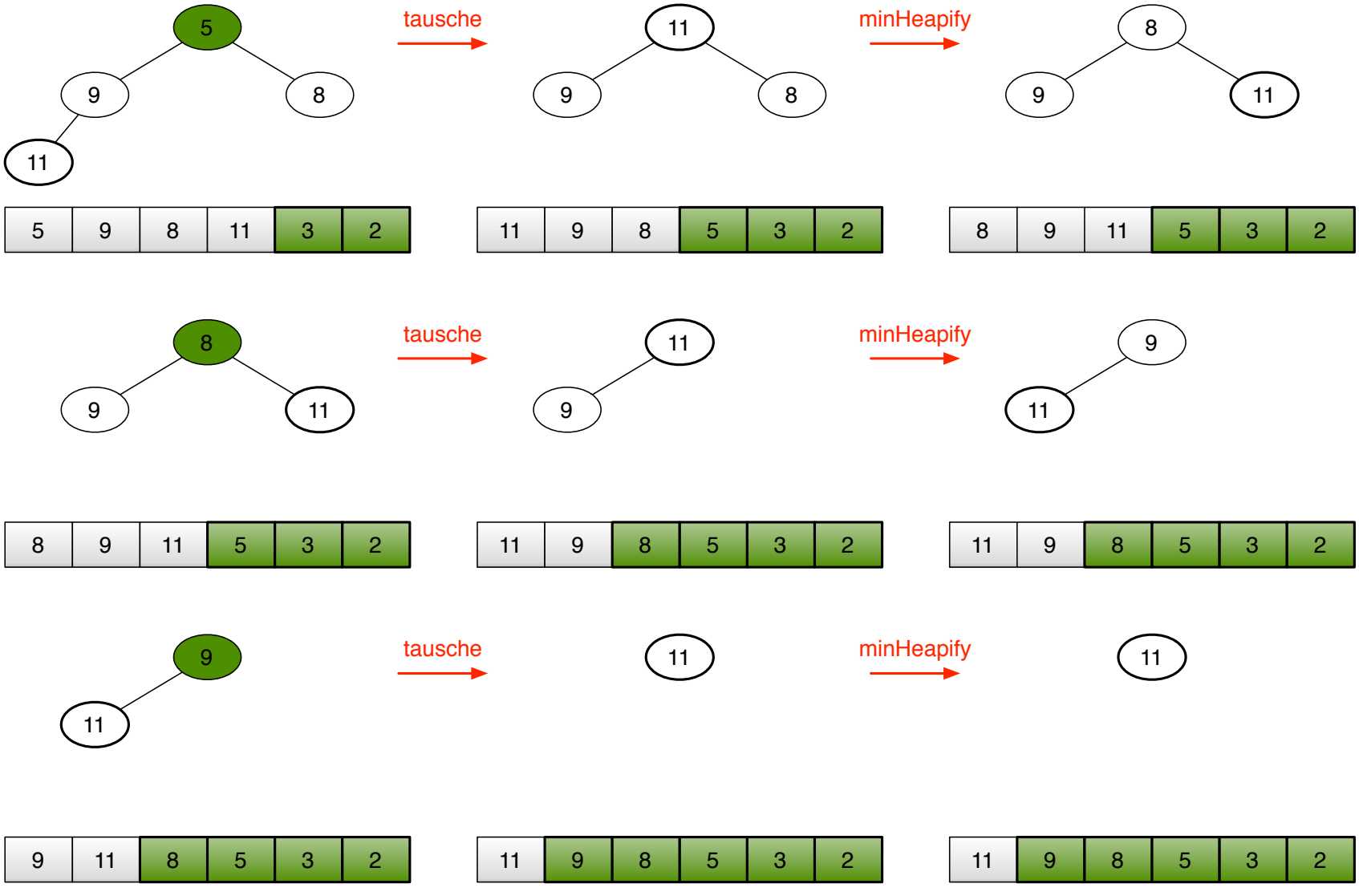
Input: Feld A[1..n] der Länge n
HeapSort(A):
    buildMinHeap(A);
    for i=n **downto** 2 {
        tausche A[1] mit A[i];
        A.length = A.length - 1;
        minHeapify(A, 1);
    }

- min-Heap sortiert in absteigender Reihenfolge
- max-Heap sortiert in aufsteigender Reihenfolge

HeapSort: Beispiel I



HeapSort: Beispiel II



HeapSort Eigenschaften

- sortiert in-place
- Komplexität $O(n \log n)$
  - besser als QuickSort im worst case!
  - in Praxis aber erst bei grossem n
- nicht stabil

Animationen der Sortier-Algorithmen:

`http://www.sorting-algorithms.com`

Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

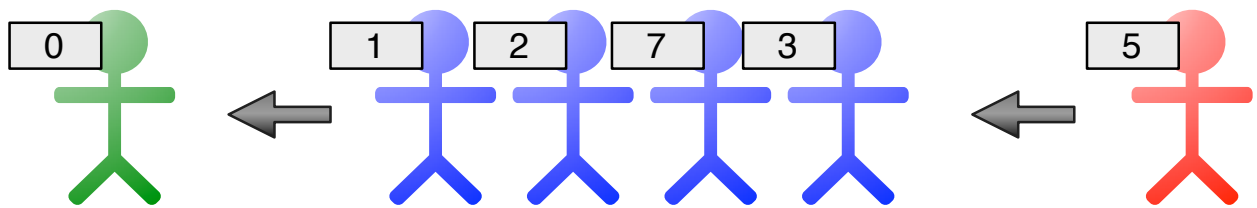
8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume

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 Elemente 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)) liefert zurück
  - falls x == minimum(insert(P, x)), dann P
  - sonst: insert(extractMin(P), x)

(entsprechend für die Priority Queue mit maximalem Schlüssel)

Priority Queue: Implementationen I

- mit sortierten Feldern (als sequentielle oder verkette Liste)
  - insert legt Element an richtiger Stelle in sortierter Liste ab mit $O(n)$ Komplexität
  - minimum, extractMin als $O(1)$ Operation
- mit unsortierten Feldern (als sequentielle oder verkettete Liste)
  - insert hängt Element einfach an Ende an mit $O(1)$
  - minimum, extractMin suchen nach Element mit kleinstem Schlüssel mit $O(n)$

$\rightarrow$ beides nicht sonderlich effizient (je nach Abfolge der Operationen aber ok)

Priority Queue: Implementationen II

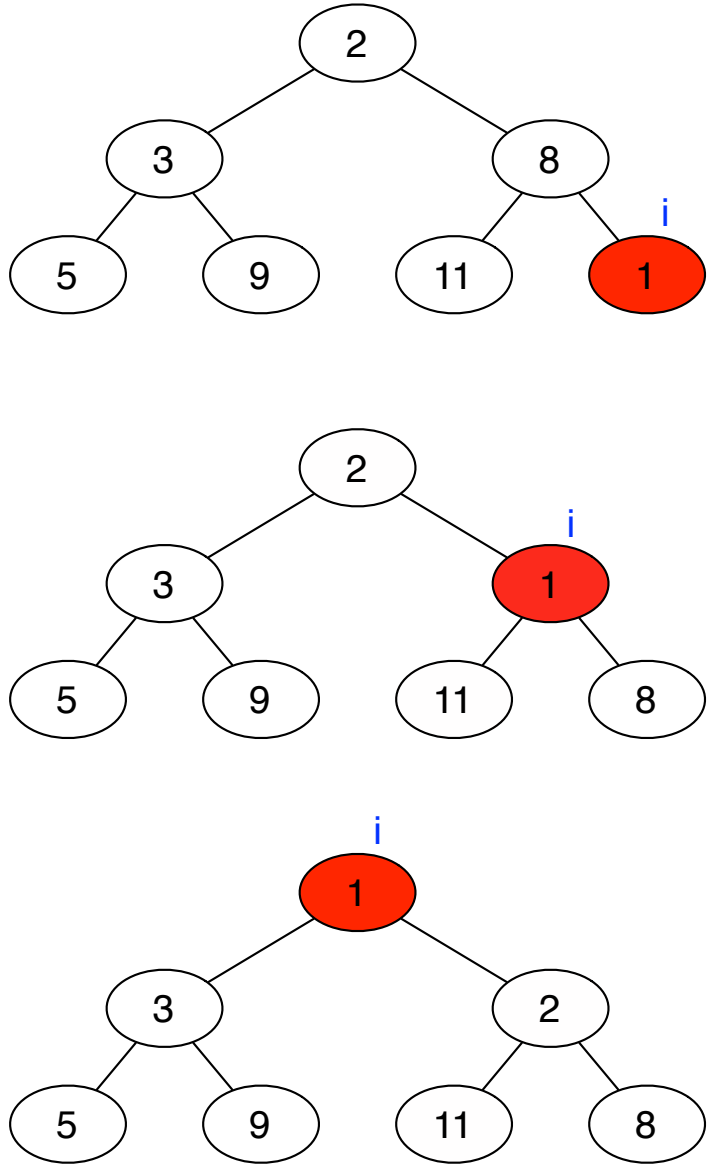
Priority Queue P als min-Heap $G = (V, E)$ mit Wurzel w:

- minimum von P liefert Wurzel w zurück
  - Komplexität $O(1)$
- extractMin von P entspricht extractMin von G
  - Komplexität $O(\log n)$
- insert von P erfordert ein klein wenig Extra-Aufwand:

  Input: Priority Queue P (als min-Heap A), Element x
  insert(A, x):
  füge Element x an Ende von Heap A ein;
  i = Index von letztem Element;
  while ((i != 1) && (A[**vater**(i)] > A[i])) {
  tausche A[i] mit A[**vater**(i)];
  i = **vater**(i);
  }

  - Komplexität $O(\log n)$

Priority Queue: insert Beispiel



Priority Queue: dynamisches Anpassen von Keys

- manchmal ändert sich der "Priorität" von Schlüsseln
  - Beispiel Algorithmen dafür in Kapitel 9!
- Operation decreaseKey verringert Schlüssel von bestimmten Element

Input: Priority Queue P (als min-Heap A), Element mit Index i, neuer Schlüsselwert wert

decreaseKey(A, i, wert):
  if (wert $>$ A[i]) **error** "neuer Schlüssel größer als alter!"
  A[i] $=$ wert;
  while ((i != 1) && (A[**vater**(i)] $>$ A[i])) {
    tausche A[i] mit A[**vater**(i)];
    i $=$ **vater**(i);
  }

- Komplexität $O(\log n)$

Priority Queue: decreaseKey / insert

- mit Operation decreaseKey läßt sich insert anders formulieren:

  Input: Priority Queue P (als min-Heap A), Element x
  insert(A, x):
  A.length = A.length + 1;
  A[A.length] = ∞;
  decreaseKey(A, A.length, x);

Priority Queue: Ausblick

- Priority Queue mit Heap: insert und decreaseKey sind $O(\log n)$
- dies läßt sich mit Fibonacci-Heap bzw. Radix-Heap verbessern auf (amortisiert) $O(1)$

  $\rightarrow$ Effiziente Algorithmen in Informatik

Priority Queues und Sortieren

- mit Priority Queues lassen sich Sortier-Algorithmen implementieren
- Schema:
  - alle Elemente in Priority Queue einfügen
  - der Reihe nach alle Elemente mit extractMin / extractMax entfernen
- Beispiele:
  - Priority Queue mit Heap: HeapSort
  - Priority Queue mit sortierter sequentieller Liste: Insertion Sort
  - Priority Queue mit unsortierter sequentieller Liste: Selection Sort

Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume

Suchen

Such-Algorithmen

Gegeben sei eine Menge M von Objekten. Ein Such-Algorithmus sucht in M nach Mustern oder nach Objekten mit bestimmten Eigenschaften.

Beispiele:

- Suche von Adresse von Person in Telefonbuch
- Suche nach Webseite mit Google Search
- Suche nach Produkt auf Amazon
- Suche nach ähnlichen Mustern: Viren-Scanner
- Suche nach Mustern: Bilderkennung
- Suche nach Tumoren in medizinischen Bildern von Patienten

Lineare Suche

Gegeben sei Array A der Länge n, das Such-Schlüssel enthält.

- einfachster Such-Algorithmus: Durchlaufen des Feldes A bis gewünschter Schlüssel gefunden
- auch genannt: Lineare Suche
- Algorithmus:

  Input: Array A[1..n] mit Schlüsseln, k gesuchter Schlüssel
  Output: Index i mit A[i] = k (sonst 0)
  linearSearch(A, k):

```
i = 1;
while ( (A[i] != k) && (i ≤ n) ) {
   i = i + 1;
}
if (i ≤ n) return i; // fündig geworden
else return 0; // nichts gefunden
```

- auch anwendbar für verkettete Listen

Lineare Suche: Komplexität

Laufzeit $T(n)$ von linearSearch:

- best-case: sofort gefunden, T(n) = 1, d.h. $T(n) = O(1)$
- worst-case: alles durchsuchen, T(n) = n, d.h. $T(n) = O(n)$
- im Mittel: Annahme jede Anordnung der Such-Schlüssel ist gleich wahrscheinlich:

$$T(n) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$$

d.h. $T(n) = O(n)$

- einfacher Algorithmus, aber nicht besonders effizient

Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume

Binäre Suche

Gegeben sei Array A der Länge n, das Such-Schlüssel enthält.

- falls häufiger gesucht wird: Array A vorsortieren! $O(n \log n)$
- Such-Algorithmus mittels Divide & Conquer
  Algorithmen-Muster:
  - Divide: vergleiche mittleres Element mit gesuchtem
  - Rekursion: falls kleiner, Rekursion auf linker Hälfte
  - Rekursion: falls grösser, Rekursion auf rechter Hälfte
  - Conquer: falls gleich, liefere Ergebnis

Binäre Suche: Algorithmus rekursiv

Input: Array A[1..n] sortierter Schlüssel, k gesuchter Schlüssel
low, high: unterer/oberer Index von aktueller Array-Hälfte
Output: Index i mit A[i] = k (sonst 0)

```
binarySearch(A, k, low, high):
    if (low > high) return 0; // nichts gefunden
    middle = ⌊(low + high) / 2⌋;
    if (A[middle] == k) return middle; // fündig geworden
    if (A[middle] > k)
       return binarySearch(A, k, low, middle-1);
    else
       return binarySearch(A, k, middle+1, high);
```

- erster Aufruf mit **binarySearch**(A, k, 1, n)

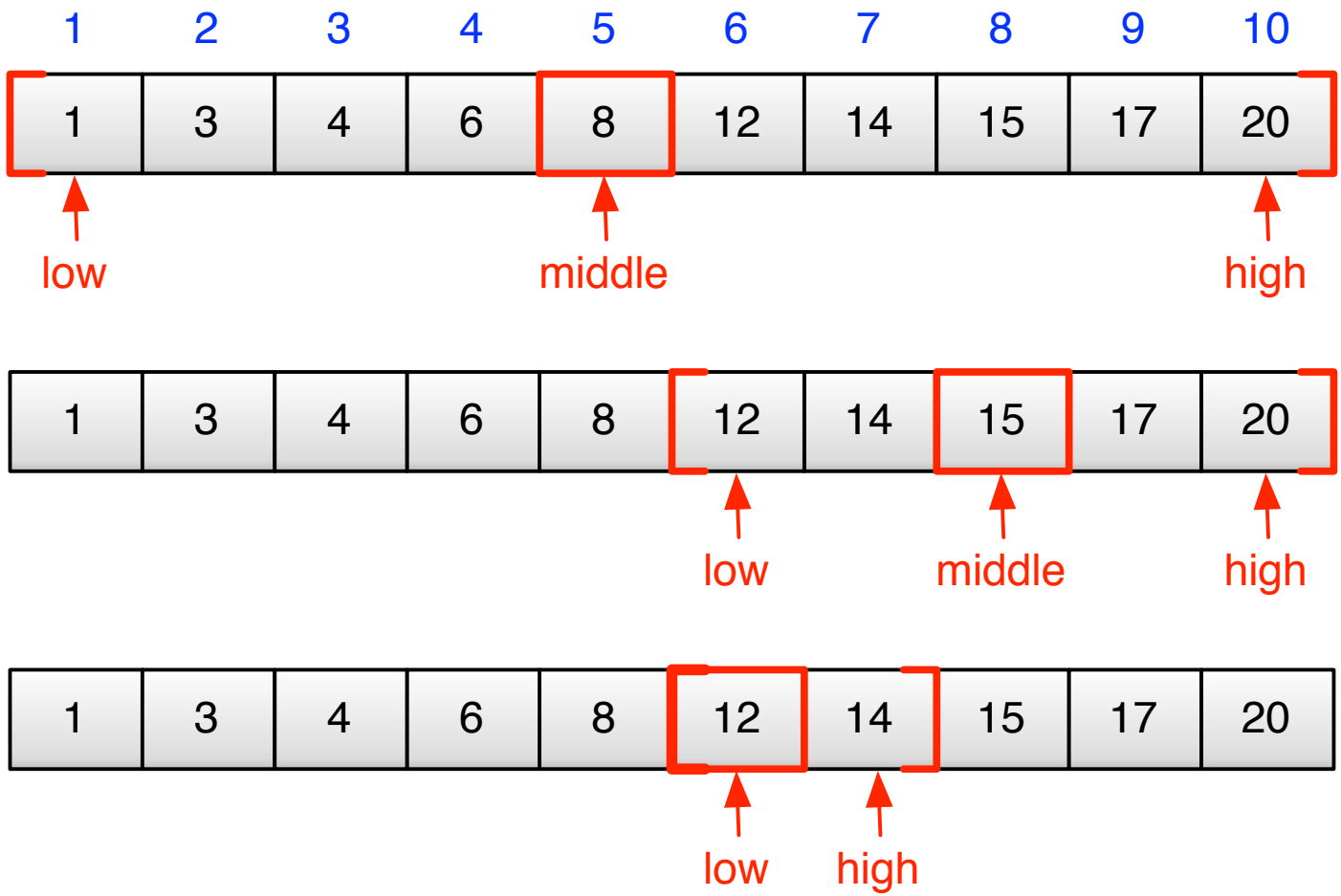
Binäre Suche: Algorithmus iterativ

Input: Array A[1..n] sortierter Schlüssel, k gesuchter Schlüssel
Output: Index i mit A[i] = k (sonst 0)

```
binarySearchIterative(A, k):
    low = 1;
    high = n;
    while (low <= high) {
        middle = ⌊ (low + high) / 2⌋;
        if (A[middle] == k) return middle; // fündig geworden
        if (A[middle] > k)
            high = middle - 1;
        else
            low = middle + 1;
    }
    return 0; // nichts gefunden
```

Binäre Suche: Beispiel

- Gesucht: Schlüssel 12



Binäre Suche: Komplexität

- Komplexität: $O(\log n)$
  - errechnet z.B. via Rekursionsbaum wie bei MergeSort
- Beispiel-Laufzeiten:

Algorithmus	$n = 10$	$n = 1000$	$n = 10^6$
Lineare Suche $(n/2)$	≈ 5	≈ 500	≈ 500.000
Binäre Suche $(\log_2 n)$	≈ 3.3	≈ 9.9	≈ 19.9

- sehr effizienter Such-Algorithmus!
- falls sich Daten oft ändern, muss jeweils neu sortiert werden
  - besser: Suchbäume

Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume

Binärer Suchbaum

Definition binärer Suchbaum

Sei $G = (V, E)$ ein Binärbaum mit Wurzel $w \in V$. Jeder Knoten $v \in V$ sei mit einem Wert $key(v)$ verknüpft, die Werte seien durch $\leq, \geq$ geordnet.
G heißt binärer Suchbaum, falls für alle inneren Knoten $v \in V$ gilt

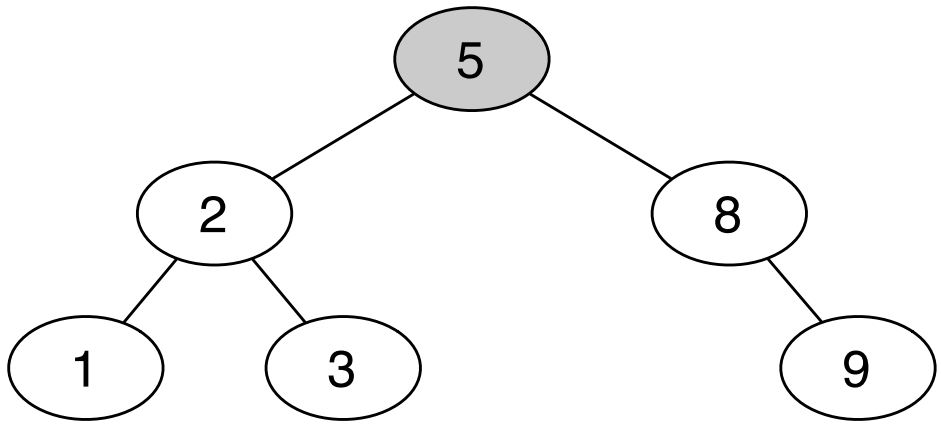
- für alle Knoten x im linken Teilbaum $v.left$ gilt

$$key(x) \leq key(v)$$

- für alle Knoten y im rechten Teilbaum $v.right$ gilt

$$key(y) \geq key(v)$$

Binärer Suchbaum: Beispiel



- binärer Baum muss nicht vollständig sein!
- Repräsentation üblicherweise mit verketteter Liste (geht aber auch als Array)

Binärer Suchbaum: Operationen

Operationen auf binärem Suchbaum:

- Suchen: finde Element mit Schlüssel k
- Minimum/Maximum: finde Element mit minimalem/ maximalem Schlüssel
- Einfügen: füge Element zum Suchbaum hinzu
- Löschen: entferne Element aus Suchbaum

Binärer Suchbaum: Suchen (rekursiv)

Input: Knoten v, dessen Teilbaum untersucht werden soll,
k gesuchter Schlüssel
Output: Knoten mit gesuchtem Schlüssel,
NULL falls nicht gefunden

```
search(v, k):
    if (v == NULL) return NULL; // hier gibt es nichts!
    if (key(v) == k) return v; // fündig geworden!
    if (k < key(v))
       search(v.left, k);
    else
       search(v.right, k);
```

- erster Aufruf mit **search**(w, k)
- falls v kein linkes/rechtes Kind hat, ist das durch NULL markiert

Binärer Suchbaum: Suchen (iterativ)

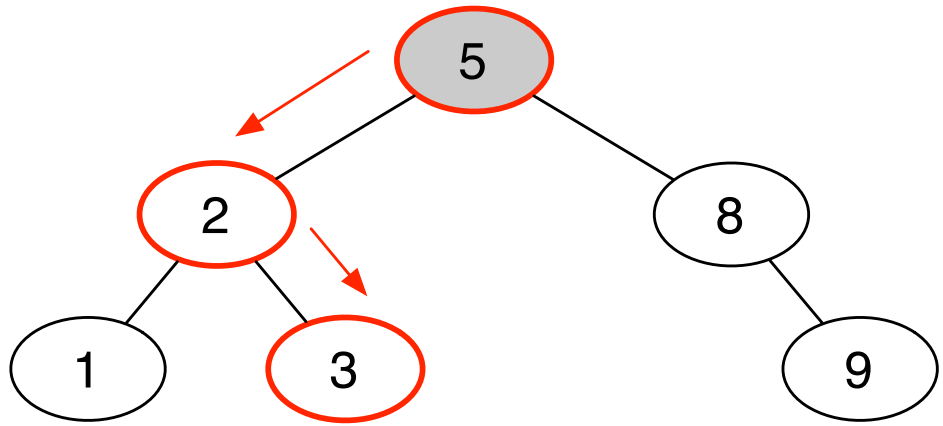
Input: Knoten v, dessen Teilbaum untersucht werden soll,
k gesuchter Schlüssel
Output: Knoten mit gesuchtem Schlüssel,
NULL falls nicht gefunden

```
searchIterative(v, k):
    while ( (v != NULL) && (key(v) != k) ) {
      if (k < key(v))
        v = v.left;
      else
        v = v.right;
    }
    return v;
```

- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

Binärer Suchbaum: Suchen

- Beispiel: suche Schlüssel 3

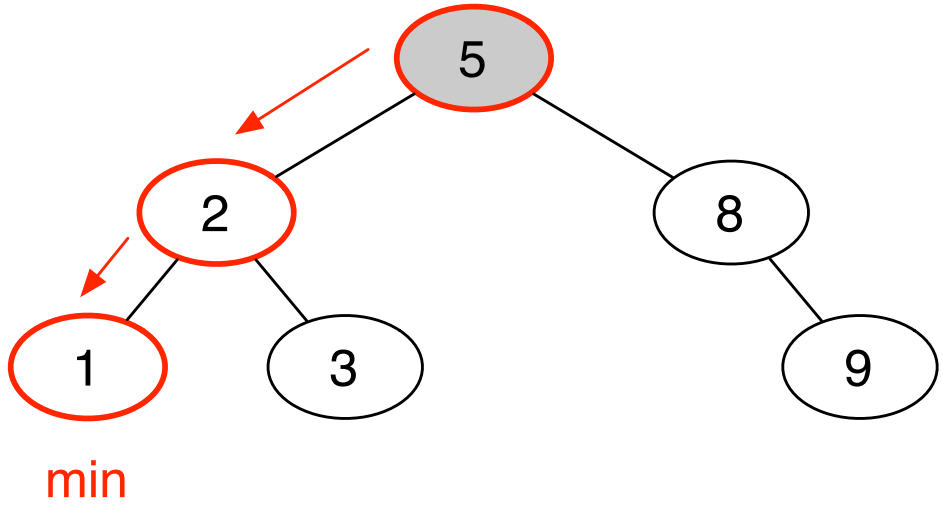


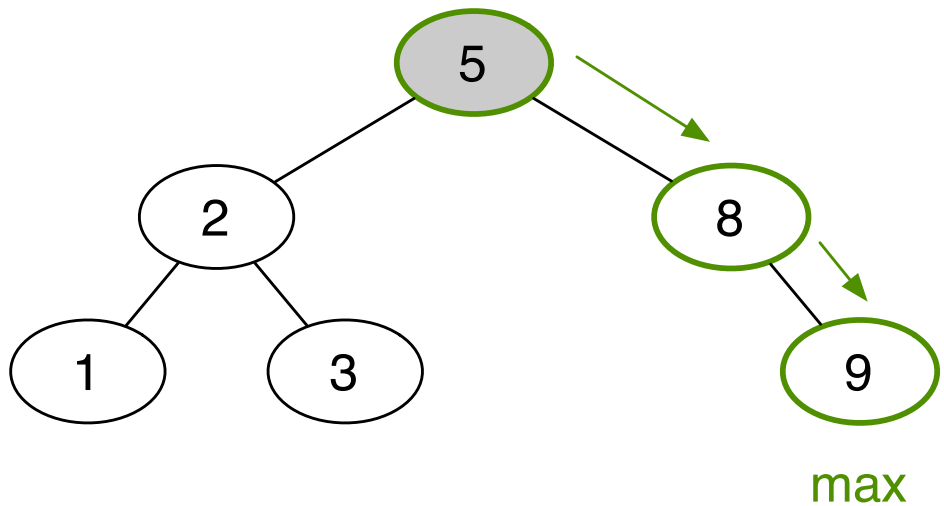
Binärer Suchbaum: Minimum/Maximum

Input: Wurzel v des zu durchsuchenden Baumes
Output: Knoten mit minimalem Schlüssel
minimum(v):
    while ($v.left$!= NULL)
        $v = v.left$;
    return v;

Input: Wurzel v des zu durchsuchenden Baumes
Output: Knoten mit maximalem Schlüssel
maximum(v):
    while ($v.right$!= NULL)
        $v = v.right$;
    return v;

Binärer Suchbaum: Minimum/Maximum





- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

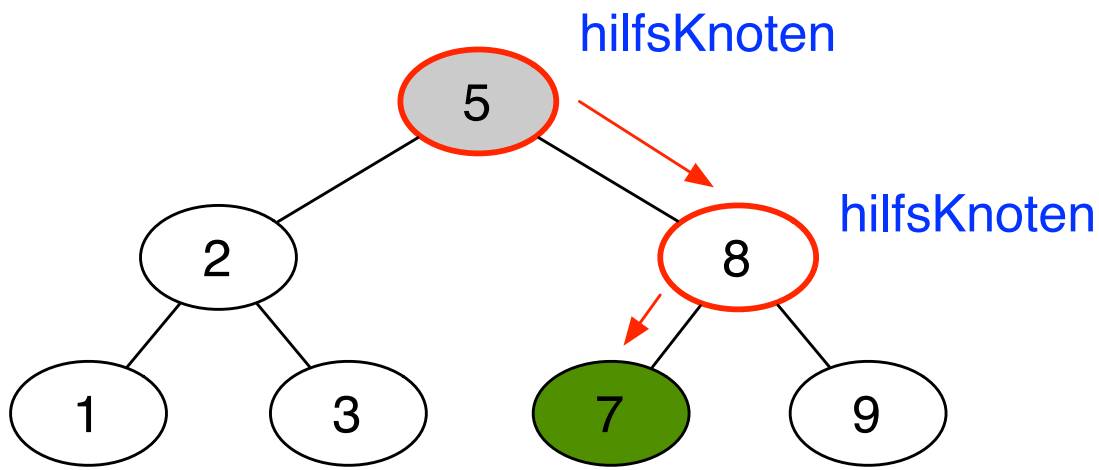
Binärer Suchbaum: Einfügen

Input: Wurzel v des Baumes, x einzufügendes Element

```
insert(v, x):
    if (v == NULL) { // Baum leer
        v = x; return;
    }
    while (v != NULL) {
        hilfsKnoten = v;
        if (key(x) < key(v))
            v = v.left;
        else
            v = v.right;
    }
    x.vater = hilfsKnoten;
    if (key(x) < key(hilfsKnoten))
        hilfsKnoten.left = x;
    else
        hilfsKnoten.right = x;
```

Binärer Suchbaum: Einfügen

- Einfügen von Knoten mit Schlüssel 7:



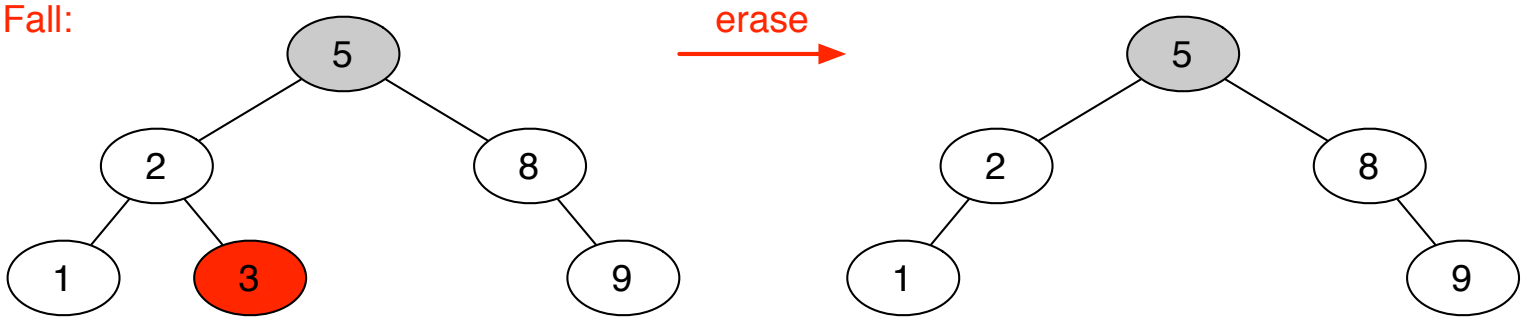
- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

Binärer Suchbaum: Löschen

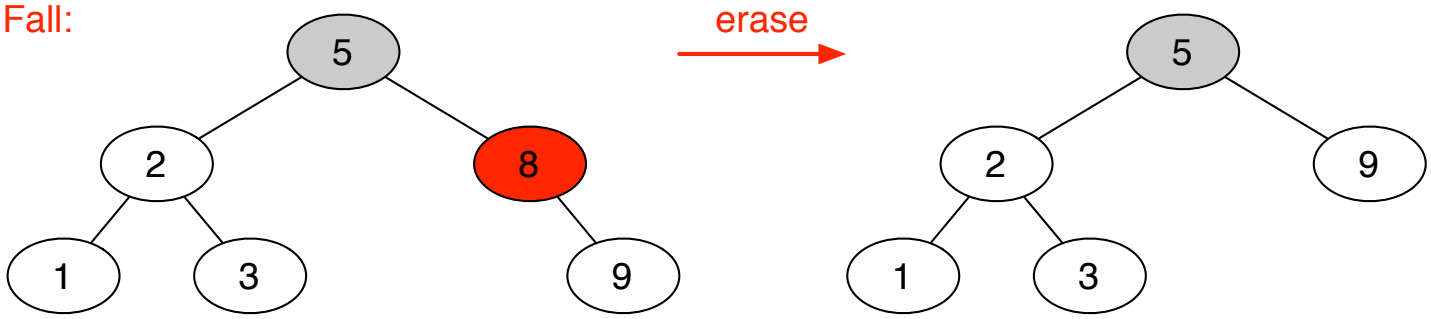
- Löschen von Knoten x in Suchbaum ist etwas komplizierter
- Drei Fälle:
  1. x ist Blatt: einfach entfernen
  2. x hat nur ein Kind: setze Kind an Stelle von x
  3. x hat zwei Kinder: setze minimales Element von rechtem Teilbaum an Stelle von x
     (alternativ: maximales Element von linkem Teilbaum)
- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

Binärerer Suchbaum: Löschen

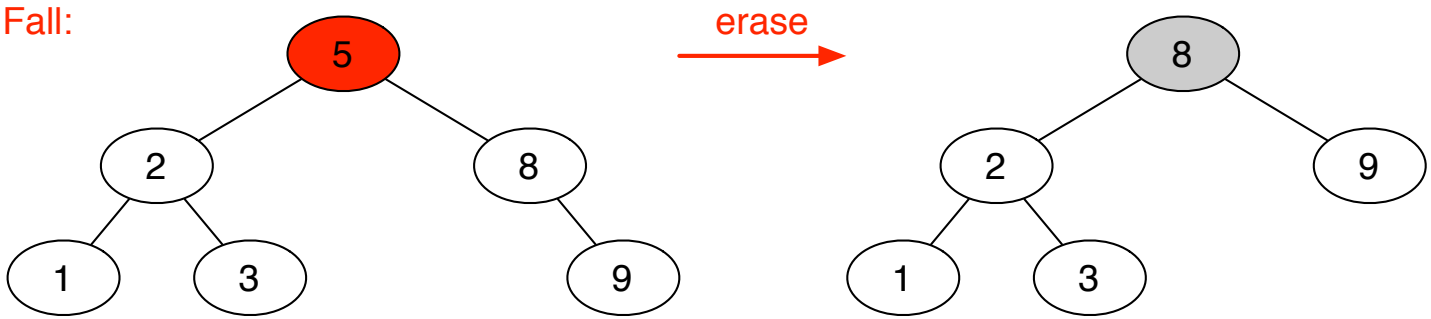
1. Fall:



2. Fall:



3. Fall:



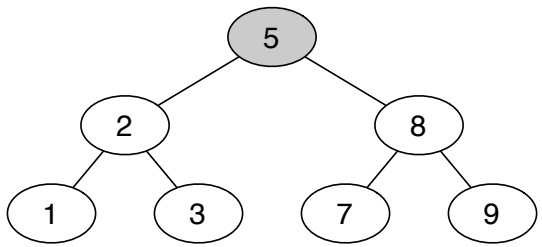
Binärer Suchbaum: Löschen

Input: Wurzel v des Baumes, x zu löschendes Element

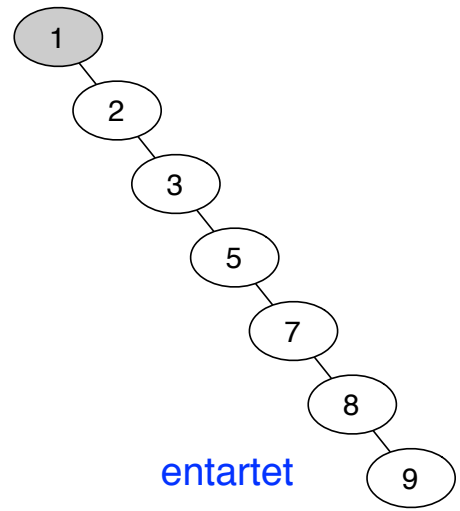
```
erase(v, x):
    if (x ist Blatt) { // 1. Fall
        if (x ist linkes Kind) x.vater.left = NULL;
        else x.vater.right = NULL;
    } else { // 2. Fall
        if (x.left == NULL) {
            if (x ist linkes Kind) x.vater.left = x.right;
            else x.vater.right = x.right;
        } else {
            if (x.right == NULL) {
                if (x ist linkes Kind) x.vater.left = x.left;
                else x.vater.right = x.left;
            } else { // 3. Fall
                kind = minimum(x.right);
                ersetze x durch kind;
            }
        }
    }
```

Binärer Suchbaum: Effizienz

- Suchbäume mit n Knoten sind sehr effizient
  - aber nur wenn sie ausgeglichen ("balanciert") sind!
- best-case Komplexität: $O(\log n)$
- worst-case Komplexität: $O(n)$



ausgeglichen



entartet

- Ausweg: automatisch balancierte Suchbäume
  (z.B. AVL Bäume, Rot-Schwarz Bäume, B-Bäume)

Zusammenfassung

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume