

Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München



Wiederholung letzte Vorlesung

DFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

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

① Pre-order Reihenfolge

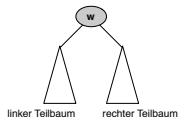
- besuche Wurzel
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

② In-order Reihenfolge

- durchlaufe linken Teilbaum
- besuche Wurzel
- durchlaufe rechten Teilbaum

③ Post-order Reihenfolge

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel



Wiederholung letzte Vorlesung

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

DFS Binär

Sei $G = ($

Tiefensuch

1 Pre-order

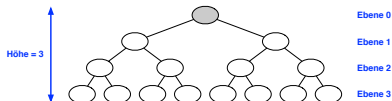
- bes
- dur
- dur

2 In-order

- dur
- bes
- dur

3 Post-order Reihentolge

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel



15

Wiederholung letzte Vorlesung

Definition Heap

Definition Heap

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 **Heap**, falls er folgende zwei Eigenschaften erfüllt:

- G ist **fast vollständig**, d.h. alle Ebenen sind vollständig gefüllt, ausser auf der untersten Ebene, die von links her nur bis zu einem bestimmten Punkt gefüllt sein muss.
- G erfüllt die **Min-Heap-Eigenschaft** bzw. die **Max-Heap-Eigenschaft**, d.h. für alle Knoten $v \in V$, $v \neq w$ gilt
 - Min-Heap: $key(v.vater) \leq key(v)$
 - Max-Heap: $key(v.vater) \geq key(v)$

Entsprechend der Heap-Eigenschaft heißt G **Min-Heap** bzw. **Max-Heap**.

BFS Binärt

Sei $G = ($

Breitensuc

- besuc
- für al
- t

Höhe =

DFS Binärt

Sei $G = ($

Tiefensuc

1 Pre-orde

- bes
- dur
- dur

2 In-order

- dur
- bes
- dur

3 Post-order Reihentolge

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel



20

15

Wiederholung letzte Vorlesung

Definition Heap

Definition Heap

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 H.

- G ist ausgebalanciert
- G erfüllt die Max-Heap Eigenschaft

Entsprechend heißt G ein Max-Heap

Heap: extractMin

Sei $G = (V, E)$ Min-Heap mit Wurzel $w \in V$.

- Operation **extractMin**:
 - entferne Wurzel w aus Heap G und liefere $key(w)$ zurück
 - tausche letzten Knoten von G an Stelle von Wurzel
 - stelle Heap-Eigenschaft wieder her mit **minHeapify**

Output: minimaler Key in G

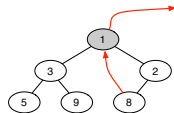
extractMin(G):

$min = key(w)$;

$w =$ letzter Knoten in G ;

minHeapify(G, w);

return min ;



BFS Binärt

Sei $G = (V, E)$

Breitensuch

- besuche alle Knoten
- für alle Kinder

DFS Binärt

Sei $G = (V, E)$

Tiefensuch

1 Pre-order

- besuche Wurzel
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

2 In-order

- durchlaufe linken Teilbaum
- besuche Wurzel
- durchlaufe rechten Teilbaum

3 Post-order Reihentolge

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

Höhe =



Wiederholung letzte Vorlesung

Definition Heap

Definition Heap

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 H.

- G ist ausgeglichen (einem Max-Heap)
- G erfüllt die Max-Heap-Eigenschaft

Entspricht einem Max-Heap

Heap: extractMin

Sei $G = (V, E)$ ein Binärbaum mit Wurzel $w \in V$ und $|V| = n$.

Heap: minHeapify

Sei $G = (V, E)$ Min-Heap mit Wurzel $w \in V$ und $|V| = n$.

- Operation **minHeapify** auf Knoten $v \in V$ zur Wiederherstellung der Min-Heap-Eigenschaft
- Voraussetzung:** nur Knoten v verletzt Min-Heap-Eigenschaft
- lasse v durch Heap absinken, bis Min-Heap-Eigenschaft wiederhergestellt

Output

Input: Knoten v

minHeapify(G, v):

```

if (v ist Blatt) return;
knoten = Minimum von v.links und v.rechts;
if (key(knoten) < key(v)) {
    tausche knoten und v;
    minHeapify(G, knoten);
}

```

- Komplexität: $O(\log n)$

BFS Binärt

Sei $G = (V, E)$

Breitensuch

- besuche die Wurzel
- für alle Kinder v von w
- für alle Kinder v von w

Höhe =



DFS Binärt

Sei $G = (V, E)$

Tiefensuch

1 Pre-order

- besuche die Wurzel
- durchlaufe links Teilbaum
- durchlaufe rechts Teilbaum

2 In-order

- durchlaufe links Teilbaum
- besuche die Wurzel
- durchlaufe rechts Teilbaum

3 Post-order Reihentolge

- durchlaufe links Teilbaum
- durchlaufe rechts Teilbaum
- besuche die Wurzel

Wiederholung letzte Vorlesung

Definition Heap

Definition Heap

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 H.

- G ist ausgelesen in einem Max-Heap
- G erfüllt

Entsprechend
Max-Heap

BFS Binärt

Sei $G = (V, E)$

Breitensuch

- besuche für alle
- t

Höhe =



Heap: extractMin

Sei $G = (V, E)$ ein Min-Heap mit Wurzel $w \in V$ und $|V| = n$.

Heap: minHeapify

- Sei $G = (V, E)$ Min-Heap mit Wurzel $w \in V$ und $|V| = n$.
- Operation $minHeapify$ auf Knoten $v \in V$ aus

Heap erzeugen: buildMinHeap

Gegeben sei Knoten-Liste V mit $|V| = n$ und Keys $key(v)$ für $v \in V$.

- wie erzeugt man aus V einen Min-Heap $G = (V, E)$?
 - erzeuge irgendwie fast vollständigen Binärbaum aus V
 - wende $minHeapify$ auf alle Knoten $v \in V$ an (nicht nötig für unterste Ebene des Baumes!)

Input: Knoten-Liste V

Output: Min-Heap G

buildMinHeap(V):

G = erzeuge beliebigen fast vollständigen Binärbaum aus V ;

for each Knoten v in G ausser unterster Ebene {

minHeapify(G, v);

}

- Komplexität: $O(n)$ (nicht nur $O(n \log n)$!)

DFS Binärt

Sei $G = (V, E)$

Tiefensuch

1 Pre-order

- besuche
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

2 In-order

- durchlaufe linken Teilbaum
- besuche
- durchlaufe rechten Teilbaum

3 Post-order Reihentolge

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

aft

Programm heute

7 Fortgeschrittene Datenstrukturen

Graphen

Bäume

Heaps

Priority Queues

8 Such-Algorithmen

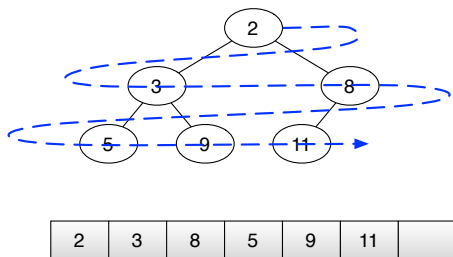
Lineare Suche

Binäre Suche

Binäre Suchbäume

Binärbaum als sequentielle Liste I

- vollständiger Binärbaum Höhe k hat $2^{k+1} - 1$ Knoten
→ speichere Knoten von oben nach unten, von links nach rechts in sequentieller Liste (Array)

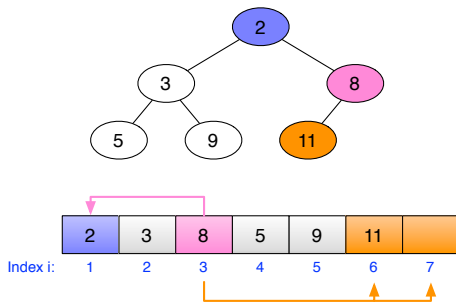


Binärbaum als sequentielle Liste II

- **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:

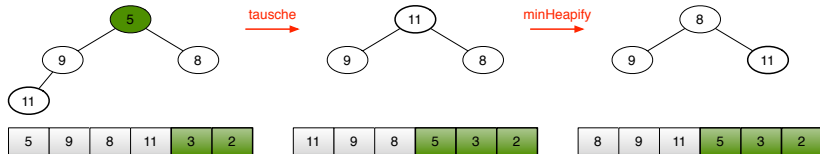
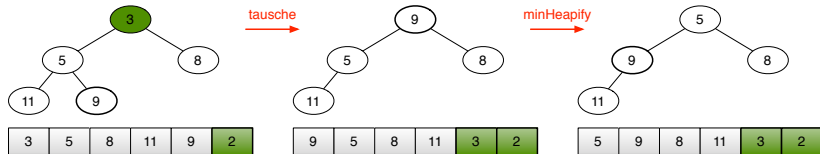
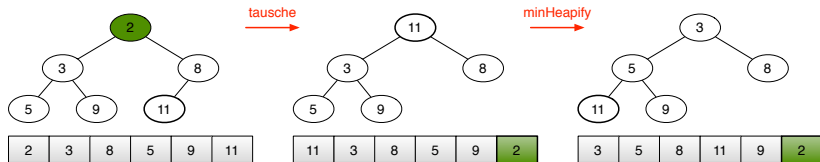
Input: Feld $A[1..n]$ der Länge n

HeapSort(A):

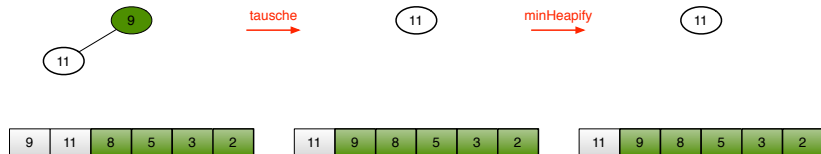
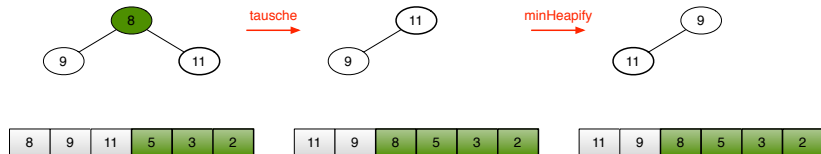
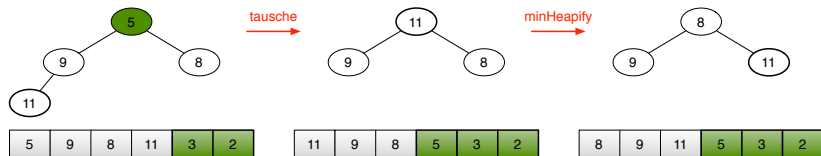
```
buildMinHeap(A);  
for  $i=n$  downto 2 {  
    tausche  $A[1]$  mit  $A[i]$ ;  
     $A.length = A.length - 1$ ;  
    minHeapify(A, 1);  
}
```

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

HeapSort: Beispiel 1



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

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

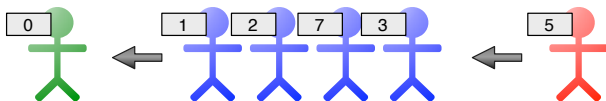
Binäre Suchbäume

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 Element 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 verkettete Liste)
 - `insert` legt Element an richtiger Stelle in sortierter Liste ab mit $O(n)$ Komplexität
 - `minimum`, `extractMin` als $O(1)$ Operation

Priority Queue: Implementationen I

- mit sortierten Feldern (als sequentielle oder verkettete 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 $O(n)$

Priority Queue: Implementationen I

- mit sortierten Feldern (als sequentielle oder verkettete 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 $O(n)$

→ 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 (($i \neq 1$) && ($A[\mathbf{vater}(i)] > A[i]$)) {

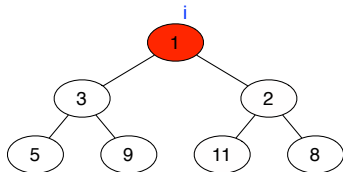
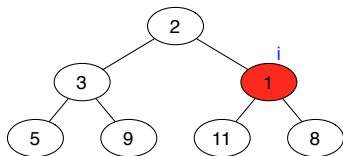
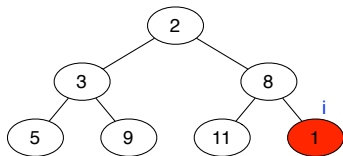
 tausche $A[i]$ mit $A[\mathbf{vater}(i)]$;

$i = \mathbf{vater}(i)$;

}

- 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 (($i \neq 1$) && ($A[vater(i)] > A[i]$)) {

 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) $O(1)$
 - 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: **Selection Sort** 
 - Priority Queue mit unsortierter sequentieller Liste: **Insertion Sort** 

Programm heute

7 Fortgeschrittene Datenstrukturen

- Graphen

- Bäume

- Heaps

- Priority Queues

8 Such-Algorithmen

- Lineare Suche

- Binäre Suche

- Binäre Suchbäume

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

5	7	3	9	11	2
---	---	---	---	----	---

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

2	3	5	7	9	11
---	---	---	---	---	----

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 = $\lfloor (low + high) / 2 \rfloor$;

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 = $\lfloor (low + high) / 2 \rfloor$;

if ($A[middle] == k$) **return** middle; // fündig geworden

if ($A[middle] > k$)

high = middle - 1;

else

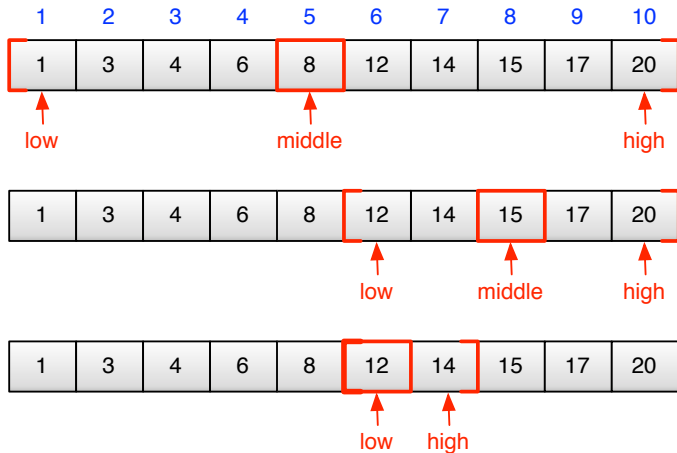
low = middle + 1;

}

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

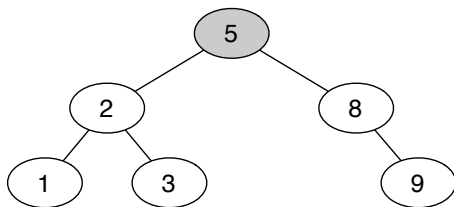
- für alle Knoten x im **linken Teilbaum** $v.left$ gilt

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

- 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 ( $v == \text{NULL}$ ) return NULL; // hier gibt es nichts!
```

```
if ( $\text{key}(v) == k$ ) return  $v$ ; // fündig geworden!
```

```
if ( $k < \text{key}(v)$ )
```

```
    search( $v.\text{left}$ ,  $k$ );
```

```
else
```

```
    search( $v.\text{right}$ ,  $k$ );
```

- **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

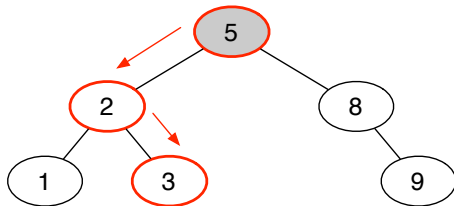
searchIterative(v, k):

```
while ( (  $v \neq \text{NULL}$  ) && (  $\text{key}(v) \neq k$  ) ) {  
    if (  $k < \text{key}(v)$  )  
         $v = v.\text{left}$ ;  
    else  
         $v = v.\text{right}$ ;  
}  
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 ( $v.left \neq \text{NULL}$ )
```

```
     $v = v.left$ ;
```

```
return  $v$ ;
```

Input: Wurzel v des zu durchsuchenden Baumes

Output: Knoten mit **maximalem** Schlüssel

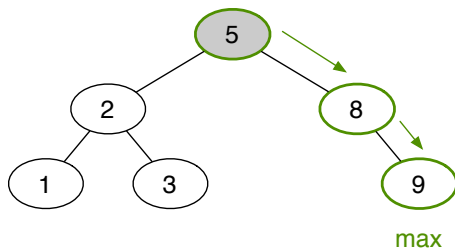
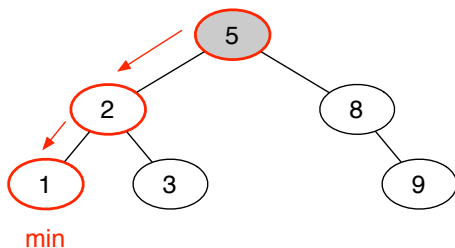
maximum(v):

```
while ( $v.right \neq \text{NULL}$ )
```

```
     $v = v.right$ ;
```

```
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 ( $v == \text{NULL}$ ) { // Baum leer
```

```
     $v = x$ ; return;
```

```
}
```

```
while ( $v \neq \text{NULL}$ ) {
```

```
    hilfsKnoten =  $v$ ;
```

```
    if ( $\text{key}(x) < \text{key}(v)$ )
```

```
         $v = v.\text{left}$ ;
```

```
    else
```

```
         $v = v.\text{right}$ ;
```

```
}
```

```
 $x.\text{vater} = \text{hilfsKnoten}$ ;
```

```
if ( $\text{key}(x) < \text{key}(\text{hilfsKnoten})$ )
```

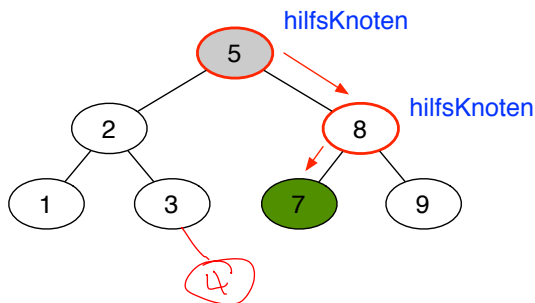
```
     $\text{hilfsKnoten}.\text{left} = x$ ;
```

```
else
```

```
     $\text{hilfsKnoten}.\text{right} = x$ ;
```


Binärer Suchbaum: Einfügen

- Einfügen von Knoten mit Schlüssel 7:



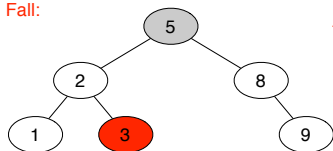
- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

Binärer Suchbaum: Löschen

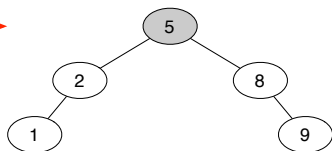
- Löschen von Knoten x in Suchbaum ist etwas komplizierter
- Drei Fälle:
 - ① x ist Blatt: einfach entfernen
 - ② x hat nur ein Kind: setze Kind an Stelle von x
 - ③ x hat zwei Kinder: setze **minimales** Element von **rechtem** Teilbaum an Stelle von x
(alternativ: maximales Element von linkem Teilbaum)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

Binärer Suchbaum: Löschen

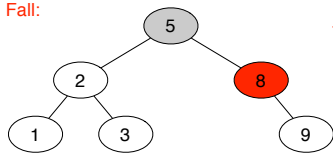
1. Fall:



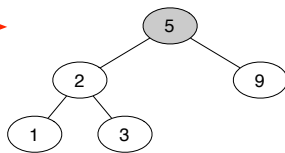
erase →



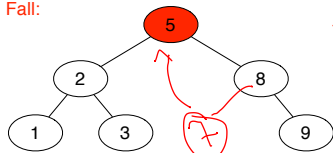
2. Fall:



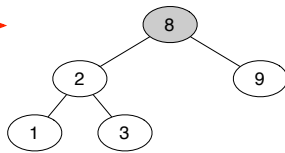
erase →



3. Fall:



erase →



Binärer Suchbaum: Löschen

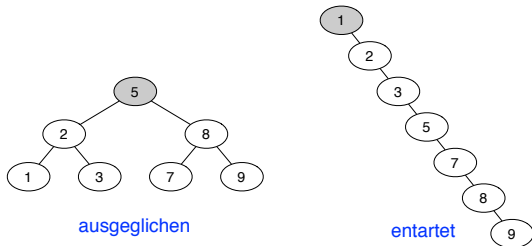
Input: Wurzel v des Baumes, x zu löschendes Element

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: $O(n)$



- 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