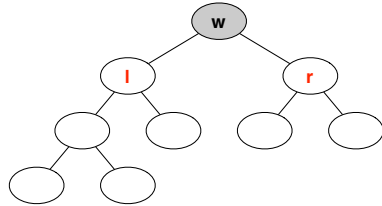


Traversierung von Binärbäumen

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 Kind-Knoten von l oder zuerst r ?
- falls r : danach Kind-Knoten von r oder zuerst l ?



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

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

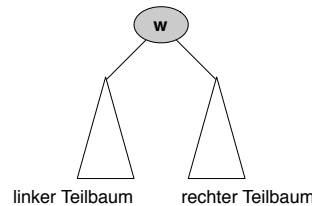
Notizen

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



Notizen

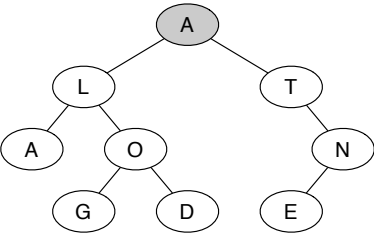
Pre-order Traversierung

Pre-order Reihenfolge:

- besuche Wurzel
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

Beispiel:

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



Notizen

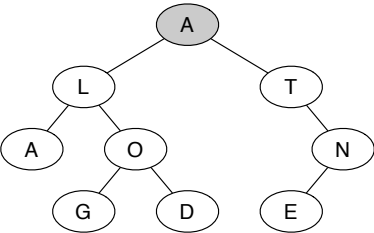
In-order Traversierung

In-order Reihenfolge:

- durchlaufe linken Teilbaum
- besuche Wurzel
- durchlaufe rechten Teilbaum

Beispiel:

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



Notizen

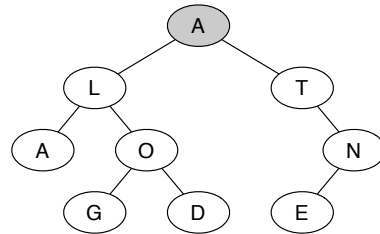
Post-order Traversierung

Post-order Reihenfolge:

- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

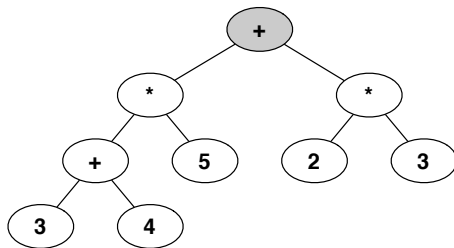
Beispiel:

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



Notizen

Beispiel: Arithmetischer Term



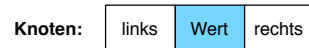
Traversierung:

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

Notizen

Implementierung DFS Traversierung

Datenstruktur:



- Algorithmus **preorder**:
preorder(knoten):
 if (knoten == NULL) **return**;
 besuche(knoten);
 preorder(knoten.links);
 preorder(knoten.rechts);
- Algorithmus **postorder**:
postorder(knoten):
 if (knoten == NULL) **return**;
 postorder(knoten.links);
 postorder(knoten.rechts);
 besuche(knoten);
- Algorithmus **inorder**:
inorder(knoten):
 if (knoten == NULL) **return**;
 inorder(knoten.links);
 besuche(knoten);
 inorder(knoten.rechts);
- rekursive Algorithmen
- auf Call-Stack basiert

Notizen

Implementierung DFS Traversierung ohne Rekursion

- Datenstruktur:



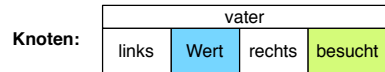
- Hilfsmittel: Stack von Knoten "knotenStack"

```
postorderIterativ(wurzelKnoten):  
  knotenStack.push(wurzelKnoten);  
  while ( !knotenStack.isEmpty() ) {  
    knoten = knotenStack.top();  
    if ( (knoten.links != NULL) && (!knoten.links.besucht) )  
      knotenStack.push(knoten.links);  
    else if ( (knoten.rechts != NULL) && (!knoten.rechts.besucht) )  
      knotenStack.push(knoten.rechts);  
    else {  
      besuche(knoten);  
      knoten.besucht = true;  
      knotenStack.pop();  
    }  
  }  
}
```

Notizen

Implementierung ohne Rekursion und ohne Stack

- Datenstruktur:



postorderIterativeOhneStack(wurzelKnoten):

```
knoten = wurzelKnoten;  
while (true) {  
  if ( (knoten.links != NULL) && (!knoten.links.besucht) )  
    knoten = knoten.links;  
  else if ( (knoten.rechts != NULL) && (!knoten.rechts.besucht) )  
    knoten = knoten.rechts;  
  else {  
    besuche(knoten);  
    knoten.besucht = true;  
    if (knoten.vater == NULL) break;  
    else knoten = knoten.vater;  
  }  
}
```

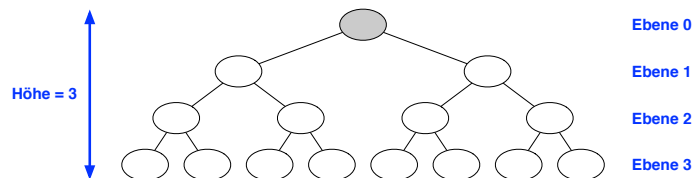
Notizen

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



Notizen

Implementierung BFS Traversierung

- Datenstruktur:



- Hilfsmittel: Queue von Knoten "knotenQueue"

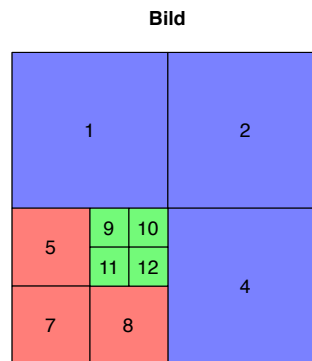
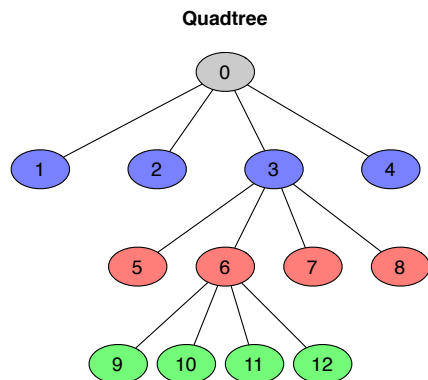
breitensuche(wurzelKnoten):

```
knotenQueue = leer;  
knotenQueue.enqueue(wurzelKnoten);  
while ( !knotenQueue.isEmpty() ) {  
    knoten = knotenQueue.dequeue();  
    besuche(knoten);  
    if (knoten.links != NULL)  
        knotenQueue.enqueue(knoten.links);  
    if (knoten.rechts != NULL)  
        knotenQueue.enqueue(knoten.rechts);  
}
```

Notizen

Anwendung: Quadtree I

- 4-närer Baum heißt **Quadtree**



Notizen

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**.

Notizen

20

Bemerkungen zur Heap Definition

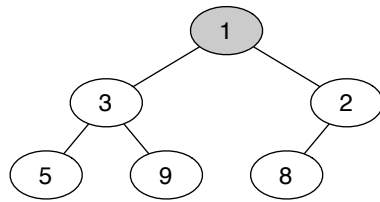
Sei $G = (V, E)$ Min-Heap.

- wir beschränken uns hier auf **Min-Heaps**, alle Aussagen gelten mit entsprechenden trivialen Änderungen auch für Max-Heaps
- typische **Keys** von Knoten sind Zahlen, z.B. $key : V \rightarrow \mathbb{R}$
 - weiteres Beispiel: Strings als Keys, lexikographisch geordnet
- ein Heap ist **kein** abstrakter Datentyp!

Notizen

21

Min-Heap: Beispiel



- Keys sind hier natürliche Zahlen
- Baum ist fast vollständiger Binärbaum
- Baum erfüllt Min-Heap-Eigenschaft:
 - für alle Knoten v (ausser Wurzel) gilt

$$\text{key}(v.\text{vater}) \leq \text{key}(v)$$

Notizen

22

Heap Eigenschaften

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

- G als Min-Heap bzw. Max-Heap hat immer Element mit kleinstem bzw. größtem Key in Wurzel w
- G ist aber nicht vollständig sortiert (d.h. Traversierung liefert nicht notwendigerweise vollständig sortierte Folge)
- Ist $|V| = n$, so hat G Höhe von $\Theta(\log n)$
- typische Operation: extrahiere kleinsten (bzw. größten) Key (d.h. Wurzel), kurz: `extractMin` (bzw. `extractMax`)
 - anschließendes Problem: Heap-Eigenschaft wiederherstellen, als Operation: `minHeapify` (bzw. `maxHeapify`)

Notizen

23

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

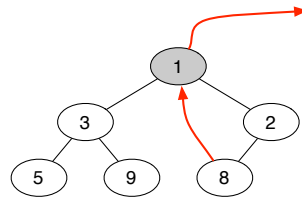
extractMin(G):

$min = key(w)$;

$w =$ letzter Knoten in G ;

minHeapify(G, w);

return min ;



Notizen

24

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

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($G, knoten$);

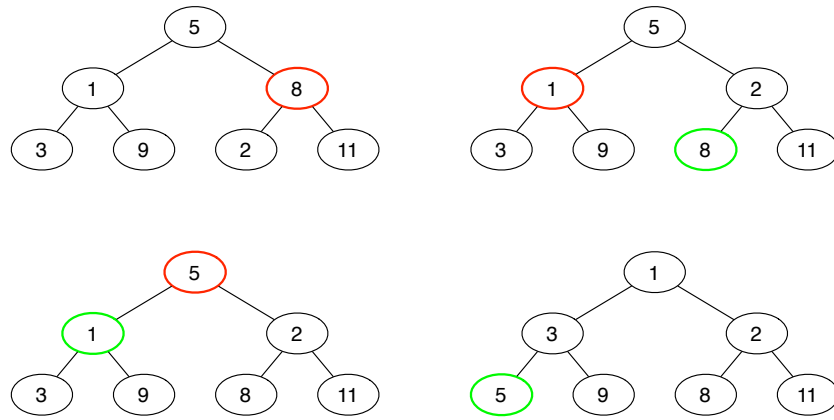
}

- Komplexität: $O(\log n)$

Notizen

25

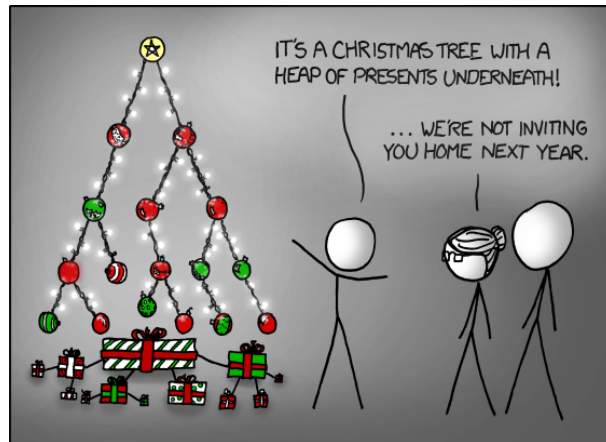
Beispiel buildMinHeap



Notizen

28

Notizen



<http://xkcd.com/835/>

29

Zusammenfassung

7 Fortgeschrittene Datenstrukturen

Graphen

Bäume

Heaps

Notizen

Notizen
