

# Datenstrukturen - Bäume und Graphen

**Dozent:** Prof. Dr. Michael Eichberg

**Kontakt:** michael.eichberg@dhbw.de, Raum 149B

**Version:** 1.1

**Quelle:** Die Folien sind teilweise inspiriert von oder basierend auf:  
Lehrmaterial von Prof. Dr. Scherer, Prof. Dr. Baumgart

---

**Folien:** <https://delors.github.io/theo-ds-baeume-und-graphen/folien.de.rst.html>  
<https://delors.github.io/theo-ds-baeume-und-graphen/folien.de.rst.html.pdf>

**Fehler melden:** <https://github.com/Delors/delors.github.io/issues>

# 1. Graphen

# Gerichteter Graph

## Definition

Ein gerichteter Graph  $G = (V, E)$  wird durch eine Knotenmenge  $V$  und eine Kantenmenge  $E$  beschrieben.

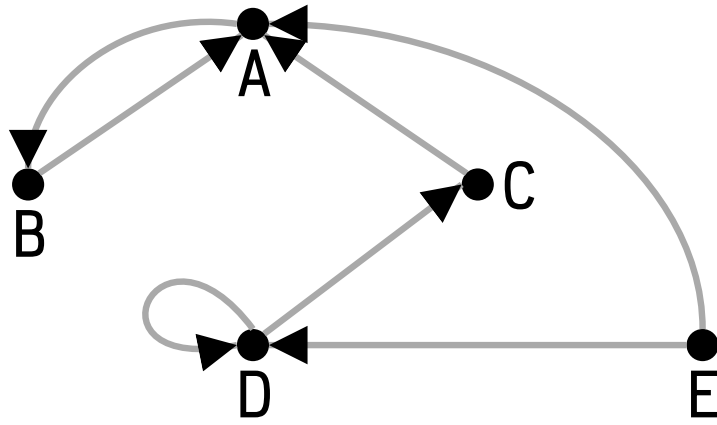
$$V = \{v_1, v_2, \dots, v_n\}$$

$$E = \{(v_i, v_j) | v_i, v_j \in V\}$$

## Bemerkung

- Es ist erlaubt, dass ein Knoten (Vertex) eine Kante (Edge) zu sich selbst besitzt
- Da  $E$  eine Menge ist, kann es keine doppelten Elemente geben; folglich gilt:
  - Es gibt maximal zwei Kanten zwischen zwei Knoten  $v_i$  und  $v_j$  ( $i \neq j$ )
  - Ein Knoten kann maximal eine Kante zu sich selbst haben
  - Statt  $v_1, \dots, v_n$  werden häufig (Groß-) Buchstaben A, B, C etc. verwendet

## Gerichteter Graph - Beispiel



# Ungerichteter Graph

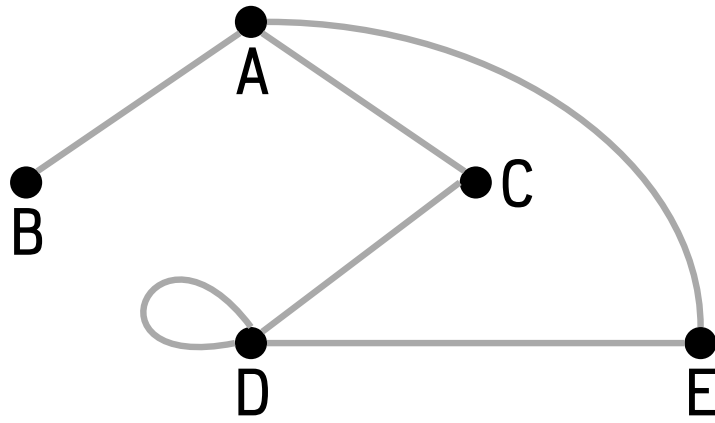
## Definition

Ein ungerichteter Graph  $G = (V, E)$  ist ein gerichteter Graph, in dem die Relation  $E$  symmetrisch ist:

$$(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$$

Die Kanten  $\{(v_i, v_j), (v_j, v_i)\}$  werden dann als eine Kante gezählt.

## Ungerichteter Graph - Beispiel



# Gewichteter Graph

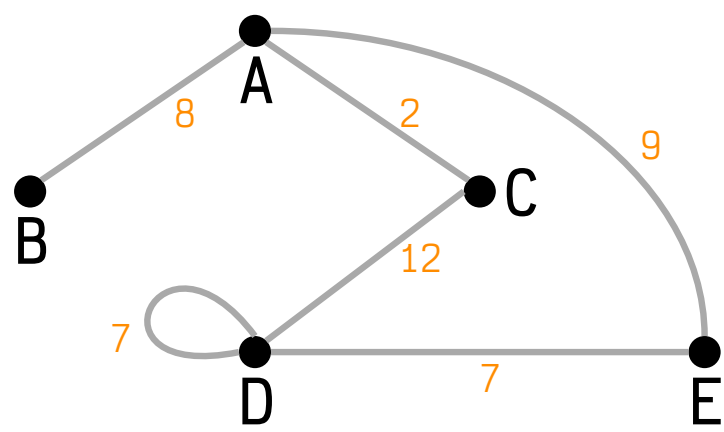
## Definition

Bei einem gewichteten Graphen  $G = (V, E)$  besitzt jede Kante  $(v_i, v_j)$  eine Attributierung  $w(v_i, v_j)$ .

## Bemerkung

- Sowohl gerichtete als auch ungerichtete Graphen können gewichtet sein
- Die Funktion  $w$  hat als Wertebereich oft  $\mathbb{N}$ ,  $\mathbb{Z}$  oder  $\mathbb{R}$
- Die Attributierung kann aber auch aus mehreren Attributen bestehen, z. B. aus einer Zahl und einer (Kanten-) Farbe; für dieses Beispiel sind die Ergebnisse von  $w$  dann Zweitupel

Gewichteter Graph - Beispiel





# Markierte Graphen

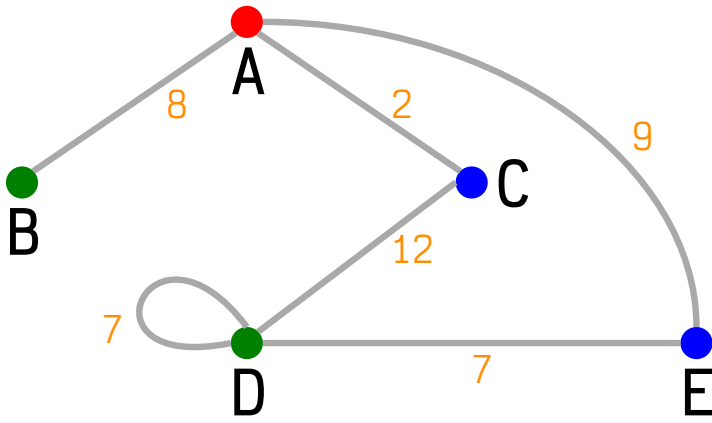
## Definition

In einem markierter Graphen  $G = (V, E)$  besitzt jeder Knoten  $v_i$  eine Markierung  $m(v_i)$ .

## Bemerkung

- Die Markierung stellt eine Attributierung des Knotens dar
- Die Markierung ist häufig ein Farbwert
- Alle Typen von Graphen (gerichtet, ungerichtet, gewichtet) können markiert werden
- Die Markierung spielt in vielen Graphenalgorithmien eine Rolle (z. B. bei der Breite-zuerst-Suche)

## Markierter (gewichteter) Graph - Beispiel



# Pfad

## Definition

Ein Pfad in einem Graphen  $G = (V, E)$  mit  $V = \{v_1, \dots, v_n\}$  ist eine Folge von Knoten  $v_{a_1}, v_{a_2}, \dots, v_{a_k}$  mit folgender Eigenschaft:  $(v_{a_i}, v_{a_{i+1}}) \in E$  für  $i \in \{1, \dots, k-1\}$ .

Die Länge des Pfades ist die Anzahl der Kanten auf dem Pfad, d. h.  $k-1$ .

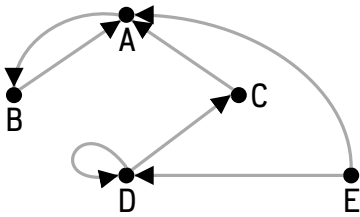
## Bemerkung

- In einem Pfad dürfen Knoten mehrfach auftreten
- Für Pfade wird häufig eine Listenschreibweise verwendet:  $\langle v_3, v_1, v_2, v_3, v_4 \rangle$

# Fragestellungen auf Graphen

- Welche Knoten sind von einem (Start-) Knoten aus erreichbar? → Erreichbarkeitsbaum
- Existiert zwischen 2 Knoten A und B ein Pfad, d. h. ist B von A aus erreichbar?
- Wenn mehrere Pfade zwischen 2 Knoten existieren, welcher ist der kürzeste?
- Existieren in einem Graphen Zyklen?

# Implementierung von Graphen



## Adjazenzliste

Struktur der Adjazenzlisten für einen Graphen  $G = (V, E)$ :

Es gibt eine Knotenliste  $L_0$ , die alle Knoten aus  $V$  enthält.

Jedem Element  $v_i (i \in v_1, \dots, v_n)$  der Knotenliste  $L_0$  ist eine weitere Knotenliste  $L_i$  zugeordnet, die die Endknoten der Kanten aus  $E$  enthält, die von  $v_i$  ausgehen

Wir speichern einen Graphen somit als Liste von Listen.

### Beispiel

Seien A bis E Instanzen der Klasse `Node`, die einen Knoten im Graphen repräsentieren.

```
// Konzeptionell
// L0 = List<Tuple<Node, List<Node>>>>(
L0 = Map<Node, List<Node>>.of(
    A, List.of(B),
    B, List.of(A),
    C, List.of(A),
    D, List.of(D, C),
    E, List.of(D, A),
)
```

## Bewertung

✓Kompaktere Speicherung

✓Erweiterbarkeit

! Aufwändigerer Zugriff z. B. bei der Adressierung der Kanten

! Höhere Komplexität für die Beantwortung von Fragen wie

■ Existiert eine Kante von x nach y?

■ Welche Kanten enden an einem Knoten x?

## Adjazenzmatrix

Struktur der Adjazenzmatrix für einen Graphen  $G = (V, E)$ :

■ Die Zeilen und Spalten der Adjazenzmatrix sind mit der Knotenmenge  $V$  beschriftet

■ Die Matrixelemente dienen zur Darstellung der Kantenrelation  $E$  eines Graphen  $G = (V, E)$

$AJ[x, y] = 1 \Leftrightarrow$  es existiert eine gerichtete Kante von x nach y - Bei einem gewichteten Graphen enthält die Adjazenzmatrix die Kantengewichte  $w(x, y)$

### Beispiel

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	0	0	0
C	1	0	0	0	0
D	0	0	1	1	0
E	1	0	0	1	0

## Bewertung

- ✓Extrem schnelle und einfache Adressierbarkeit der Kanten
- ✓Semantik einer Zeile: Welche Kanten gehen von dem zugehörigen Knoten weg
- ✓Semantik einer Spalte: Welche Kanten enden an dem zugehörigen Knoten
- ! Bei wenigen Kanten ist die Matrix nur dünn besetzt → Speicherplatzverschwendung
- ! Arrays als typische Implementierung für Matrizen erfordern Reorganisationsaufwand, wenn dem Graphen neue Knoten hinzugefügt werden



# Suchalgorithmen auf Graphen - Grundlegendes

## Eingaben

- (Gerichteter) Graph  $G = (V, E)$
- Startknoten  $v_s$

## Ablauf

- S1: Markiere Startknoten  $v_s$
- S2: Wähle Kante  $(v_i, v_j)$  mit folgenden Eigenschaften:
  - $v_i$  ist markiert und  $v_j$  ist nicht markiert
- S3: Markiere  $v_j$  und setze mit Schritt S2 fort

## Endebedingung

Es existiert in Schritt S2 kein unmarkierter Knoten mehr

## Ergebnis

Genau ein Erreichbarkeitsbaum wird markiert.

## Wichtige Varianten

- Breite-zuerst-Suche
- Tiefe-zuerst-Suche

(Die Unterscheidung erfolgt im Schritt S2 in Hinblick auf die auszuwählende Kante.)

## Benötigte Konzepte

- Aktiver und passiver Knoten
  - Kennzeichnung eines Knotens als aktiv, wenn von ihm aus die Suche (gegebenenfalls) später fortgesetzt werden kann.
- allg. Vorgehensweise
  - Kennzeichnung eines neu markierten Knotens als aktiv
  - Ein aktiver Knoten  $v$  wird passiv, wenn gilt:
    - Alle von  $v$  ausgehenden Kanten wurden dahingehend untersucht, ob sie zu einem noch nichtmarkierten Knoten führen.

## Ergebnis

- Weitersuche ist nur von aktiven Knoten aus sinnvoll
- Nur markierte Knoten können aktiv sein



# Breite-zuerst-Suche (🇺🇸 *Breadth-first-Search*)

## Im Schritt S2

- Durchsuchung aller vom Suchknoten ausgehenden Kanten dahingehend, ob sie zu einem (bisher) unmarkierten Knoten führen.
- Markierung eines derartigen Knotens.
- Aufnahme des Knotens in die Liste aktiver Knoten.
- Gibt es keine derartige Kante mehr, wähle als neuen Suchknoten den ältesten noch aktiven Knoten.

(Als Datenstruktur dient die Verwendung einer FIFO-Warteschlange)

## Endekriterium

Vom Suchknoten aus wird kein unmarkierter Nachfolgerknoten mehr gefunden und die Liste aktiver Knoten ist leer.

### !! Wichtig

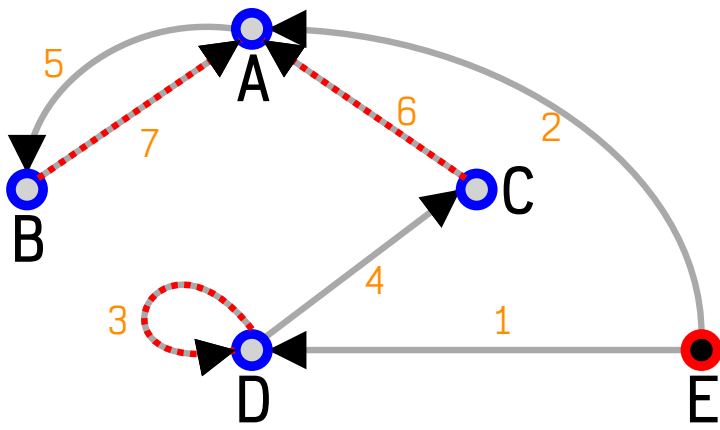
Unter allen möglichen Erreichbarkeitsbäumen wird derjenige mit den kürzesten Pfaden erzeugt.

Dies liegt an der Struktur der Suchsteuerung in S2:

- Inkrementeller Aufbau der Pfade
- Weiterhin werden die Pfade nicht sequentiell, sondern simultan aufgebaut

⇒ Knoten werden markiert, bevor lange Pfade entstehen können

## Breite-zuerst-Suche - Beispiel



# Tiefe-zuerst-Suche

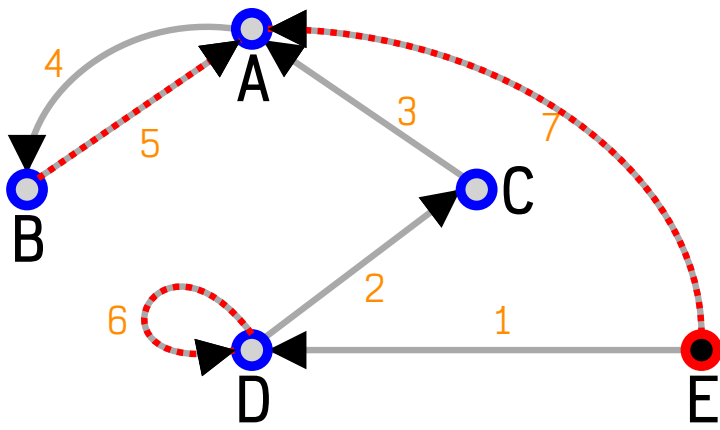
## Im Schritt S2

- Suche vom aktuellen Suchknoten  $nd$  eine Kante zu einem (bisher) unmarkierten Knoten  $nd_{neu}$
- Markierung von  $nd_{neu}$
- Aufnahme des Knotens  $nd$  (nicht  $nd_{neu}$ !) in die Liste aktiver Knoten
- $nd_{neu}$  wird zum neuen Suchknoten und die Suche wird mit  $nd_{neu}$  fortgesetzt
- Gibt es keine derartige Kante mehr, wähle als neuen Suchknoten den jüngsten noch aktiven Knoten  
(Zu verwendende Datenstruktur: Stack (LiFO))

## Endekriterium

Vom Suchknoten aus wird kein unmarkierter Nachfolgerknoten mehr gefunden und die Liste aktiver Knoten ist leer.

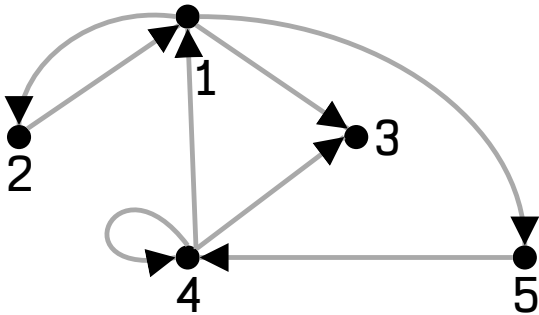
## Tiefe-zuerst-Suche - Beispiel



# Übung

## 1.1. Tiefe-/Breite-zuerst-Suche

Gegeben sei der folgenden Graph:



Führen Sie sowohl eine Tiefe- als auch Breite-zuerst-Suche ausgehend von dem Knoten 4 als auch Knoten 5 durch. Sollte es mehrere Möglichkeiten geben, dann wählen Sie zuerst den Knoten mit der kleineren Nummer!

## 2. Bäume

# Bäume - Einführung

- Bäume sind eine Datenstruktur für hierarchische Daten

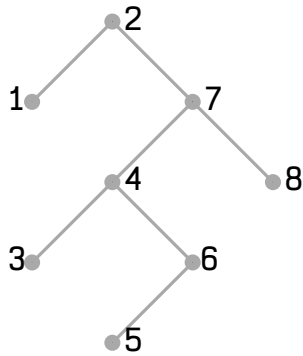
## Beispiele

- Stammbaum
- Dateisystem
- Organisationsstruktur
- relevante Operationen
  - Schnelles Abändern
  - Schnelles Suchen

## Bäume - Schlüsselbegriffe



## Beispielbaum



### Elternknoten (Parent)

Ein Elternknoten ist ein Knoten, der einen oder mehrere Kinder hat.

#### Beispiel

2 ist der Elternknoten von 7

### Kindknoten (Child(ren))

Kindknoten sind Knoten, die einen Elternknoten haben.

#### Beispiel

1 und 7 sind Kindknoten von 2

### Wurzel (Root)

Der Wurzelknoten ist der einzige Knoten, der keinen Elternknoten hat.

#### Beispiel

2 ist die Wurzel/der Wurzelknoten

### Blattknoten (Leaf)

Ein Blattknoten ist ein Knoten, der keine Kinder hat.

#### Beispiel

8, 3, 5 sind Blattknoten

### Innerer Knoten (Inner Node)

Ein Knoten, der kein Blattknoten ist, ist ein innerer Knoten.

#### Beispiel

4, 7 und 6 sind innere Knoten. Auch der Wurzelknoten ist ein innerer Knoten, wenn der Baum nicht degeneriert ist.

Die Höhe eines Baumes ist die Länge des längsten Pfades vom Wurzelknoten zu einem Blattknoten.

Die Tiefe eines Knotens  $p$  ist die Länge des Pfades vom Wurzelknoten zu  $p$ .

## Binärbaum

### Definition

Ein Binärbaum ist ein Baum, bei dem jeder Knoten höchstens zwei Kinder hat.

## Binärer Suchbaum

### Definition

Ein binärer Suchbaum ist ein Binärbaum, bei dem für jeden Knoten alle Schlüssel in den linken Unterbäumen kleiner sind und alle Schlüssel in den rechten Unterbäumen größer sind.

#### Beispiel

(siehe Darstellung)

### Achtung!

Da die Kanten in einem Baum grundsätzlich von einem Eltern- zu einem Kindknoten gehen (d. h. es sind gerichtete Kanten), wird die Kantenrichtung typischerweise nicht angegeben (d. h. es wird keine Pfeilspitze gezeichnet).



# Traversierung von Bäumen

Traversierung von Bäumen beschreibt die Reihenfolge, in der Knoten eines Baumes besucht werden.

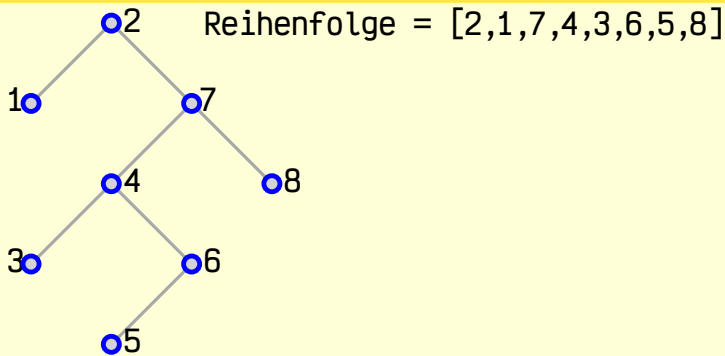
Wir können drei Formen von Traversierung unterscheiden:

1. Präorder
2. Inorder und
3. Postorder

Traversierung.

Präorder Traversierung (🚩 *Preorder Traversal*)

Beispiel

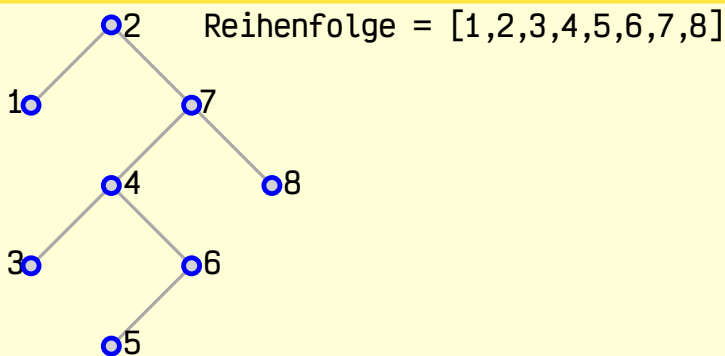


Definition

Eine Präorder Traversierung eines (Teil-)Baumes beginnt mit der Verarbeitung des aktuellen (Wurzel-)Knotens, danach besuchen wir rekursiv den linken Unterbaum, bevor der rechte Unterbaum besucht wird.

Inorder Traversierung (🚩 *Inorder Traversal*)

Beispiel



Definition

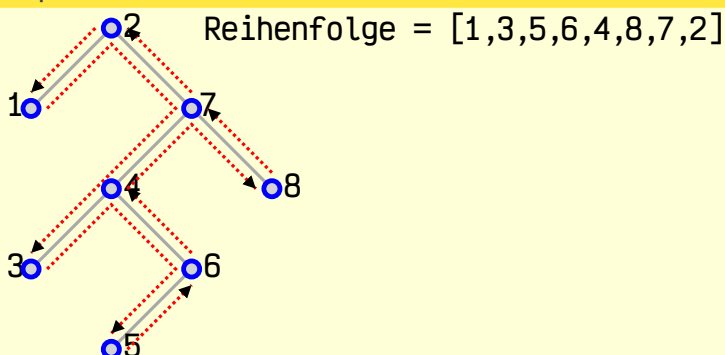
Eine Inorder-Traversierung eines Baumes beginnt mit den linken Unterbaum, besucht dann den (Wurzel-)knoten, bevor der rechte Unterbaum besucht wird.

Bemerkung

Eine Inorder-Traversierung eines (binären) Suchbaums führt zu einer sortierten Liste der Knotenwerte. Es handelt sich somit um ein weiteres Beispiel für ein Sortierverfahren.

Postorder Traversierung (🚩 *Postorder Traversal*)

Beispiel



Definition

Eine Postorder-Traversierung eines Baumes beginnt mit den linken Unterbaum und besucht dann den rechten Unterbaum, bevor der (Wurzel-)knoten besucht wird.

# Implementierung eines binären Suchbaums (BST)

```
1 // BST.java
2
3 import java.util.Deque;
4 import java.util.LinkedList;
5 import java.util.Queue;
6
7 public class BST<Key extends Comparable<Key>, Value> {
8
9     private class Node {
10         private Key key;
11         private Value val;
12         private Node left, right;
13         private int count;
14
15         public Node(Key key, Value val) {
16             this.key = key;
17             this.val = val;
18             this.count = 1;
19         }
20     }
21
22     private Node root;
23
24     public void put(Key key, Value val) {
25         root = put(root, key, val);
26     }
27
28     private Node put(Node x, Key key, Value val) {
29         if (x == null)
30             return new Node(key, val);
31         int cmp = key.compareTo(x.key);
32         if (cmp < 0)
33             x.left = put(x.left, key, val);
34         else if (cmp > 0)
35             x.right = put(x.right, key, val);
36         else if (cmp == 0)
37             x.val = val;
38         x.count = 1 + size(x.left) + size(x.right);
39
40         return x;
41     }
42
43     public Value get(Key key) {
44         Node x = root;
45         while (x != null) {
46             int cmp = key.compareTo(x.key);
47             if (cmp < 0)
48                 x = x.left;
49             else if (cmp > 0)
50                 x = x.right;
51             else if (cmp == 0)
52                 return x.val;
53         }
54         return null;
55     }
56
57     public int size() {
```

```

58     return size(root);
59 }
60
61 private int size(Node x) {
62     if (x == null)
63         return 0;
64     return x.count;
65 }
66
67 public Key select(int k) { // returns the nth-largest key
68     if (k < 0)         return null;
69     if (k ≥ size())    return null;
70     Node x = select(root, k);
71     return x.key;
72 }
73
74 private Node select(Node x, int k) {
75     if (x == null) return (Node) null;
76     int t = size(x.left);
77     if (t > k)
78         return select(x.left, k);
79     else if (t < k)
80         return select(x.right, k - t - 1);
81     else // if (t == k)
82         return x;
83 }
84
85 public void delete(Key key) {
86     /* root = */ delete(root, key);
87 }
88
89 private Node delete(Node x, Key key) {
90     if (x == null)    return null;
91     int cmp = key.compareTo(x.key);
92     if (cmp < 0)      x.left = delete(x.left, key);
93     else if (cmp > 0) x.right = delete(x.right, key);
94     else {
95         if (x.right == null)    return x.left;
96         if (x.left == null)     return x.right;
97
98         Node t = x;
99         x = min(t.right);
100        x.right = deleteMin(t.right);
101        x.left = t.left;
102    }
103    x.count = size(x.left) + size(x.right) + 1;
104    return x;
105 }
106 }

```

# Übung

## 2.1. Optimale Einfügereihenfolge

Nehmen wir an, dass wir im Voraus schätzen können, wie oft auf Suchschlüssel in einem binären Baum zugegriffen wird. Sollten die Schlüssel in wachsender oder fallender Reihenfolge der zu erwartenden Zugriffshäufigkeit in den Baum eingefügt werden? Warum?

---

## 2.2. Minimum bestimmen

Implementieren Sie die Methode `min` (`public Node min() {...}`) als rekursive Methode.

### 2.3. Rückgabe in absteigender Reihenfolge

Implementieren Sie eine Methode `public Iterable<Key> descending(){... }`, die die Schlüssel in umgekehrter Reihenfolge zurückgibt (d. h. der größte Schlüssel zuerst, der kleinste Schlüssel zuletzt). Implementieren Sie diese Methode rekursiv.

#### Bemerkung

Es ist eine gute Hausübung die Methode auch einmal nicht-rekursiv zu implementieren.