

# Tiefensuche, Breitensuche

## Tiefesuche

### Weiterführende Literatur:

- Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6, Seite 39-52 (PDF 32-45)
- Wikipedia-Artikel „Tiefensuche“
- Schneider, Taschenbuch der Informatik, Kapitel 6.2.2.2 Graphalgorithmen, Seite 185

Die Tiefensuche (englisch *depth-first search*, *DFS*) ist in der Informatik ein Verfahren zum Suchen von Knoten in einem Graphen.

Der Tiefendurchlauf ist das Standardverfahren zum Durchlaufen eines Graphen bei dem *jeder Knoten mindestens einmal* und *jede Kante genau einmal* besucht wird. Man geht vom jeweiligen Knoten *erst zu einem nicht besuchten Nachbarknoten* und setzt den Algorithmus dort *rekursiv* fort. Bei schon besuchten Knoten wird abgebrochen. Als Hilfsstruktur wird ein *Stack* (Stapelspeicher, Keller) verwendet. Der konkrete Durchlauf hängt von der Reihenfolge der Knoten in den Adjazenzlisten bzw. in der Adjazenzmatrix ab.<sup>1</sup>

depth-first search

DFS

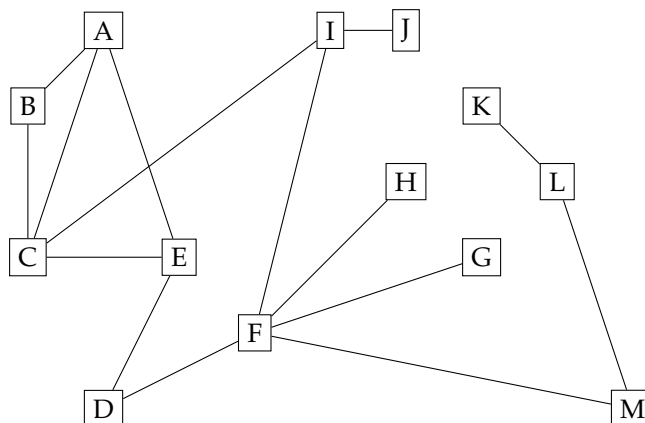
jeder Knoten mindestens einmal

jede Kante genau einmal

erst zu einem nicht besuchten Nachbarknoten

rekursiv

Stack



Startknoten A:

**A:**

**Besuchte Knoten:** A, B

**Stack:** A

**B:**

**Besuchte Knoten:** A, B, C

**Stack:** A, B

<sup>1</sup>Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6, Seite 40.

**C:**

**Besuchte Knoten:** A, B, C, E

**Stack:** A, B, C

**B:** A, B, C

**Stack:** A, B, C

**C:** A, B, C, E

**Stack:** A, B, C, E

**E:** A, B, C, E, D

**Stack:** A, B, C, E

**E:** A und C schon markiert A, B, C, E, D

**Stack:** A, B, C, E, D

**D:** A, B, C, E, D, F

**Stack:** A, B, C, E, D

**F:** A, B, C, E, D, F, M

**Stack:** A, B, C, E, D, F

**M:** A, B, C, E, D, F, M, L

**Stack:** A, B, C, E, D, F, M

**L:** A, B, C, E, D, F, M, L, K

**Stack:** A, B, C, E, D, F, M, L

**K:** keine Kinder A, B, C, E, D, F, M, L, K

**Stack:** A, B, C, E, D, F, M, L

**L:** alle Kinder schon markiert A, B, C, E, D, F, M, L, K

**Stack:** A, B, C, E, D, F, M

**M:** alle Kinder schon markiert A, B, C, E, D, F, M, L, K

**Stack:** A, B, C, E, D, F

**F:** A, B, C, E, D, F, M, L, K, G

**Stack:** A, B, C, E, D, F

**G:** keine Kinder

**F:** A, B, C, E, D, F, M, L, K, G, H

**Stack:** A, B, C, E, D, F

**H:** keine Kinder

**F:** A, B, C, E, D, F, M, L, K, G, H, I

**Stack:** A, B, C, E, D, F

**I:** A, B, C, E, D, F, M, L, K, G, H, I, J

**Stack:** A, B, C, E, D, F, I

**J:** keine Kinder I: alle Kinder markiert F: alle Kinder markiert D: alle Kinder markiert E: alle Kinder markiert C: alle Kinder markiert B: alle Kinder markiert A: alle Kinder markiert

A, B, C, E, D, F, M, L, K, G, H, I, J

**Stack:** leer

ENDE

## Implementierung der Tiefensuche

- Eine Möglichkeit, abzuspeichern, welche Knoten bereits besucht wurden  
→ Boolean-Array
- Eine Methode, die für uns diese Markierung der Knoten als besucht übernimmt (und somit die eigentliche Tiefensuche durchführt) → Knoten als besucht eintragen, existierende Nachbarknoten suchen und prüfen, ob diese bereits besucht wurden, falls nicht: diese durch rekursiven Aufruf besuchen
- Eine Methode, um die Tiefensuche zu starten → Wenn ein übergebener Startknoten existiert, dann müssen erst alle Knoten als nicht besucht markiert werden und dann vom Startknoten aus das Besuchen der Knoten gestartet werden

2

## Pseudocode: Tiefensuche mit explizitem Stack<sup>3</sup>

### Algorithmus 1: Tiefensuche mit explizitem Stack

```

Data:  $G$ : Graph,  $k$ : Startknoten in  $G$ 
 $S :=$  leerer Stack;
lege  $k$  oben auf  $S$ ;
markiere  $k$ ;
while  $S$  nicht leer ist do
     $a :=$  entferne oberstes Element von  $S$ ;
    bearbeite Knoten  $a$ ;
    for alle Nachfolger  $n$  von  $a$  do
        if  $n$  noch nicht markiert then
            lege  $n$  oben auf  $S$ ;
            markiere  $n$ ;
        end
    end
end

```

<sup>2</sup>Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6, Seite 45.

<sup>3</sup>Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6, Seite 51 (PDF 45).

```

3  import org.bsclangaul.graph.GraphAdjazenzMatrix;
4  import org.bsclangaul.helfer.Farbe;
5
6  import java.util.ArrayList;
7  import java.util.List;
8  import java.util.Stack;
9  import java.util.Vector;
10
11  /**
12   * nach Schulbuch: Informatik 1 Oberstufe Oldenbourg Verlag
13   */
14  public class TiefenSucheStapel extends GraphAdjazenzMatrix {
15
16      /**
17       * Der Schnappschuss wird entweder erstellt, nachdem ein Knoten besucht wurde,
18       * oder ein Knoten aus dem Stapel entfernt wurde.
19       */
20      class SchnappSchuss {
21          String besuchterKnoten;
22          String entnommenerKnoten;
23
24          public SchnappSchuss(Stack<String> stapel) {
25              this.kopiereStapel(stapel);
26          }
27
28          /**
29           * Eine Kopie des referenzierten Stapels als einfaches Feld.
30           */
31          Object[] stapel;
32
33          void kopiereStapel(Stack<String> stapel) {
34              this.stapel = stapel.toArray();
35          }
36
37          SchnappSchuss merkeBesuch(String knotenName) {
38              this.besuchterKnoten = knotenName;
39              return this;
40          }
41
42          SchnappSchuss merkeEntnahme(String knotenName) {
43              this.entnommenerKnoten = knotenName;
44              return this;
45          }
46      }
47
48      class Protokoll {
49          List<SchnappSchuss> schnappSchuesse;
50
51          /**
52           * Eine Referenze auf den vom Algorithmus verwendeten Stapel.
53           */
54          Stack<String> stapel;
55
56          public Protokoll(Stack<String> stapel) {
57              this.schnappSchuesse = new ArrayList<SchnappSchuss>();
58              this.stapel = stapel;
59          }
60
61          void merkeBesuch(String knotenName) {
62              schnappSchuesse.add(new SchnappSchuss(stapel).merkeBesuch(knotenName));
63          }
64

```

```
65     void merkeEntnahme(String knotenName) {
66         schnappSchuesse.add(new SchnappSchuss(stapel).merkeEntnahme(knotenName));
67     }
68 }
69
70 /**
71  * Liste der besuchten Knoten
72  */
73 private boolean[] besucht;
74
75 /**
76  * Stapel für die Tiefensuche
77  */
78 private Stack<String> stapel = new Stack<String>();
79 private Vector<String> route = new Vector<String>();
80
81 Protokoll protokoll = new Protokoll(stapel);
82
83 /**
84  * Die maximale Anzahl der Knoten wird dabei festgelegt.
85  *
86  * @param maximaleKnoten Anzahl der maximal möglichen Knoten
87  */
88 public TiefenSucheStapel(int maximaleKnoten) {
89     super(maximaleKnoten);
90     initialisiereTiefensuche(maximaleKnoten);
91 }
92
93 /**
94  * Die Adjazenzmatrix kann mit diesem Konstruktor im einfachen Graphenformat
95  * spezifiziert werden.
96  *
97  * @param einfachesGraphenFormat Ein String im einfachen Graphenformat.
98  */
99 public TiefenSucheStapel(String einfachesGraphenFormat) {
100     super(einfachesGraphenFormat);
101     initialisiereTiefensuche(gibKnotenAnzahl());
102 }
103
104 private void initialisiereTiefensuche(int maximaleKnoten) {
105     besucht = new boolean[maximaleKnoten];
106     route = new Stack<String>();
107 }
108
109 public void besuche(int knotenNummer) {
110     String name = gibKnotenName(knotenNummer);
111     besucht[knotenNummer] = true;
112     route.add(name);
113     stapel.push(name);
114     protokoll.merkeBesuch(name);
115     System.out.println(Farbe.rot("besucht: ") + name);
116     System.out.println(Farbe.grün("Stapel: ") + stapel.toString());
117 }
118
119 /**
120  * Durchlauf aller Knoten und Ausgabe auf der Konsole
121  *
122  * @param knotenNummer Nummer des Startknotens
123  */
124 public void besucheKnoten(int knotenNummer) {
125     besuche(knotenNummer);
126     // Stapel ausgeben
```

```

127 while (!stapel.isEmpty()) {
128     // oberstes Element des Stapels nehmen und in die Route einfügen
129     String knotenName = stapel.pop();
130     protokoll.merkeEntnahme(knotenName);
131     System.out.println(Farbe.gelb("Aus dem Stapel entfernen: ") + knotenName);
132
133     // alle nicht besuchten Nachbarn von w in den Stapel einfügen
134     for (int abzweigung = 0; abzweigung <= gibKnotenAnzahl() - 1; abzweigung++)
135         ↪ {
136             if (matrix[gibKnotenNummer(knotenName)][abzweigung] != NICHT_ERREICHBAR
137                 ↪ && !besucht[abzweigung]) {
138                 besuche(abzweigung);
139             }
140         }
141     // Route ausgeben
142     System.out.println("\n" + Farbe.gelb("Route: ") + route.toString());
143 }
144
145 /**
146  * Start der Tiefensuche
147  * @param startKnoten Bezeichnung des Startknotens
148  */
149 public void führeAus(String startKnoten) {
150     int startnummer;
151     startnummer = gibKnotenNummer(startKnoten);
152
153     if (startnummer != -1) {
154         for (int i = 0; i <= gibKnotenAnzahl() - 1; i++) {
155             besucht[i] = false;
156         }
157         besucheKnoten(startnummer);
158     }
159 }
160
161 public static void main(String[] args) {
162     TiefenSucheStapel ts = new TiefenSucheStapel(20);
163
164     ts.setzeKnoten("A");
165     ts.setzeKnoten("B");
166     ts.setzeKnoten("C");
167     ts.setzeKnoten("D");
168
169     ts.setzeKnoten("E");
170     ts.setzeKnoten("F");
171     ts.setzeKnoten("G");
172     ts.setzeKnoten("H");
173     ts.setzeKnoten("J");
174     ts.setzeKnoten("K");
175
176     ts.setzeUngerichteteKante("A", "B", 1);
177     ts.setzeUngerichteteKante("A", "C", 1);
178
179     ts.setzeUngerichteteKante("B", "A", 1);
180     ts.setzeUngerichteteKante("B", "D", 1);
181     ts.setzeUngerichteteKante("B", "E", 1);
182
183     ts.setzeUngerichteteKante("C", "A", 1);
184     ts.setzeUngerichteteKante("C", "F", 1);
185     ts.setzeUngerichteteKante("C", "G", 1);
186

```

```

187     ts.setzeUngerichteteKante("D", "B", 1);
188     ts.setzeUngerichteteKante("D", "H", 1);
189
190     ts.setzeUngerichteteKante("E", "B", 1);
191     ts.setzeUngerichteteKante("E", "F", 1);
192
193     ts.setzeUngerichteteKante("F", "C", 1);
194     ts.setzeUngerichteteKante("F", "E", 1);
195     ts.setzeUngerichteteKante("F", "G", 1);
196     ts.setzeUngerichteteKante("F", "J", 1);
197
198     ts.setzeUngerichteteKante("G", "C", 1);
199     ts.setzeUngerichteteKante("G", "F", 1);
200
201     ts.setzeUngerichteteKante("H", "D", 1);
202
203     ts.setzeUngerichteteKante("J", "F", 1);
204
205     ts.setzeUngerichteteKante("K", "F", 1);
206
207     ts.gibMatrixAus();
208
209     ts.führeAus("A");
210 }
211 }

```

Code-Beispiel auf Github ansehen: [src/main/java/org/bschlangaul/graph/algorithmen/TiefenSucheStapel.java](https://github.com/src/main/java/org/bschlangaul/graph/algorithmen/TiefenSucheStapel.java)

## Breitensuche

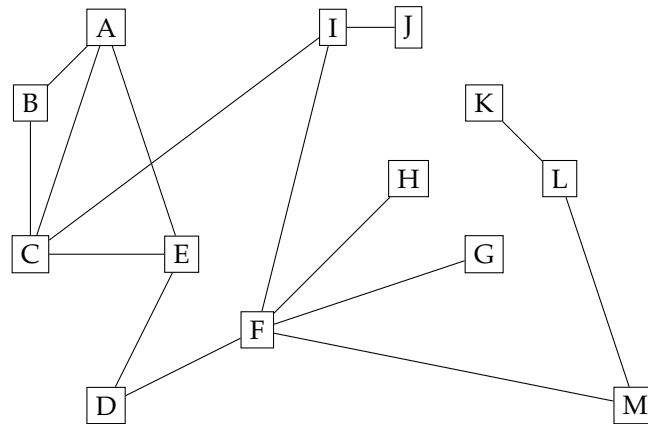
### Weiterführende Literatur:

- *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6*, Seite 53-64 (PDF 46-57)
- Schneider, *Taschenbuch der Informatik*, Kapitel 6.2.2.2 Graphalgorithmen, Seite 185
- *Wikipedia-Artikel „Breitensuche“*

Der Breitendurchlauf (englisch *breadth-first search*, *BFS*)<sup>4</sup> ist Verfahren zum Durchlaufen eines Graphen bei dem jeder Knoten genau einmal besucht wird. Man geht von einem Knoten *erst zu allen Nachbarknoten bevor deren Nachbarn* besucht werden. Bei schon besuchten Knoten wird abgebrochen. Als Hilfsstruktur wird eine *Queue* (*Warteschlange*) verwendet.

Der konkrete Durchlauf hängt von der Reihenfolge der Knoten in den Adjazenzlisten ab.

<sup>4</sup>Wikipedia-Artikel „Breitensuche“.



**A:** A, B, C, E   **w:** B, C, E

**B:** alle Kinder markiert

A, B, C, E w: C, E

**C:** A, B und E markiert

A, B, C, E, I w: E, I

**E:** A und C markiert

A, B, C, E, I, D w: I, D

I: C markiert

A, B, C, E, I, D, F, J w: D, F, J

**D:** E, F sind markiert A, B, C, E, I, D, F, J w: F, J

**F:** I ist markiert A, B, C, E, I, D, F, J, M w: J, M, G, H

J: keine Kinder

A, B, C, E, I, D, F, J, M, G, H w: M, G, H

**M:** A, B, C, E, I, D, F, J, M, G, H, L **w:** G, H, L

**G:** keine Kinder A, B, C, E, I, D, F, J, M, G, H, L w: H, L

**H:** keine Kinder A, B, C, E, I, D, F, J, M, G, H, L w: L

L: A, B, C, E, I, D, F, J, M, G, H, L, K w: K

**K:** keine Kinder A, B, C, E, I, D, F, J, M, G, H, L, K w:

ENDE



## Pseudocode Breitensuche mit Queue<sup>5</sup>

### Algorithmus 2: Breitensuche mit Queue

```

Data:  $G$ : Graph,  $k$ : Startknoten in  $G$ 
 $Q$  := leerer Queue;
füge  $k$  in  $Q$ ;
markiere  $k$ ;
while  $Q$  nicht leer do
     $a$  := entferne vorderstes Element aus  $Q$ ;
    bearbeite Knoten  $a$ ;
    for alle Nachfolger  $n$  von  $a$  do
        if  $n$  noch nicht markiert then
            füge  $n$  hinten in  $Q$ ;
            markiere  $n$ ;
        end
    end
end

```

```

3  import org.bschlangaul.graph.GraphAdjazenzMatrix;
4  import org.bschlangaul.helfer.Farbe;
5
6  import java.util.ArrayList;
7  import java.util.List;
8  import java.util.Vector;
9
10 /**
11  * nach Schulbuch Informatik 1 Oberstufe Oldenbourg Verlag
12  */
13 public class BreitenSucheWarteschlange extends GraphAdjazenzMatrix {
14
15     /**
16     * Der Schnappschuss wird entweder erstellt, nachdem ein Knoten besucht wurde,
17     * oder ein Knoten aus dem Stapel entfernt wurde.
18     */
19     class SchnappSchuss {
20         String besuchterKnoten;
21         String entnommenerKnoten;
22
23         public SchnappSchuss(Vector<String> warteschlange) {
24             this.kopiereStapel(warteschlange);
25         }
26
27         /**
28         * Eine Kopie des referenzierten Stapels als einfaches Feld.
29         */
30         Object[] warteschlange;
31
32         void kopiereStapel(Vector<String> warteschlange) {
33             this.warteschlange = warteschlange.toArray();
34         }
35
36         SchnappSchuss merkeBesuch(String knotenName) {
37             this.besuchterKnoten = knotenName;
38             return this;

```

<sup>5</sup>Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6, Seite 64.

```

39     }
40
41     SchnappSchuss merkeEntnahme(String knotenName) {
42         this.entnommenerKnoten = knotenName;
43         return this;
44     }
45 }
46
47 class Protokoll {
48     List<SchnappSchuss> schnappSchuesse;
49
50     /**
51      * Eine Referenz auf den vom Algorithmus verwendeten Stapel.
52      */
53     Vector<String> warteschlange;
54
55     public Protokoll(Vector<String> warteschlange) {
56         this.schnappSchuesse = new ArrayList<SchnappSchuss>();
57         this.warteschlange = warteschlange;
58     }
59
60     void merkeBesuch(String knotenName) {
61         schnappSchuesse.add(new
62             ↳ SchnappSchuss(warteschlange).merkeBesuch(knotenName));
63     }
64
65     void merkeEntnahme(String knotenName) {
66         schnappSchuesse.add(new
67             ↳ SchnappSchuss(warteschlange).merkeEntnahme(knotenName));
68     }
69 }
70
71 /**
72  * Liste der besuchten Knoten
73  */
74 private boolean[] besucht;
75
76 /**
77  * Eine Warteschlange für die Breitensuche
78  */
79 private Vector<String> warteschlange = new Vector<String>();
80 private Vector<String> route = new Vector<String>();
81
82 Protokoll protokoll = new Protokoll(warteschlange);
83
84 /**
85  * Die Adjazenzmatrix kann mit diesem Konstruktor im einfachen Graphenformat
86  * spezifiziert werden.
87  *
88  * @param einfachesGraphenFormat Ein String im einfachen Graphenformat.
89  */
90 public BreitenSucheWarteschlange(String einfachesGraphenFormat) {
91     super(einfachesGraphenFormat);
92     besucht = new boolean[gibKnotenAnzahl()];
93 }
94
95 public void besuche(int knotenNummer) {
96     String name = gibKnotenName(knotenNummer);
97     besucht[knotenNummer] = true;
98     route.add(name);
99     warteschlange.add(name);
100     protokoll.merkeBesuch(name);

```

```

99     System.out.println(Farbe.rot("besucht: ") + name);
100     System.out.println(Farbe.grün("Warteschlange: ") + warteschlange.toString());
101 }
102
103 /**
104  * Durchlauf aller Knoten und Ausgabe auf der Konsole
105  *
106  * @param knotenNummer Nummer des Startknotens
107  */
108 private void besucheKnoten(int knotenNummer) {
109     besuche(knotenNummer);
110
111     while (!warteschlange.isEmpty()) {
112         // oberstes Element der Liste nehmen
113         String knotenName = warteschlange.remove(0);
114         System.out.println(Farbe.gelb("Aus der Warteschlange entfernen: ") +
115             ↪ knotenName);
116         protokoll.merkeEntnahme(knotenName);
117
118         // alle nicht besuchten Nachbarn von knotenName in die Liste einfügen
119         for (int abzweigung = 0; abzweigung <= gibKnotenAnzahl() - 1; abzweigung++)
120             ↪ {
121             if (matrix[gibKnotenNummer(knotenName)][abzweigung] != NICHT_ERREICHBAR
122                 ↪ && !besucht[abzweigung]) {
123                 besuche(abzweigung);
124             }
125         }
126     }
127     // Route ausgeben
128     System.out.println(Farbe.gelb("Route: ") + route.toString());
129 }
130
131 /**
132  * Start der Breitensuche
133  *
134  * @param startKnoten Bezeichnung des Startknotens
135  */
136 public void führeAus(String startKnoten) {
137     int startnummer;
138     startnummer = gibKnotenNummer(startKnoten);
139
140     if (startnummer != -1) {
141         for (int i = 0; i <= gibKnotenAnzahl() - 1; i++) {
142             besucht[i] = false;
143         }
144         besucheKnoten(startnummer);
145     }
146 }
147
148 public static void main(String[] args) {
149     BreitenSucheWarteschlange bs = new BreitenSucheWarteschlange(
150         ↪ "a--e; a--f; a--s; b--c; b--d; b--h; c--d; c--h; c--s; d--h; e--f; f--s; g--s; h--s;");
151     bs.gibMatrixAus();
152     bs.führeAus("s");
153 }

```

Code-Beispiel auf Github ansehen: [src/main/java/org/bschlangaul/graph/algorithmen/BreitenSucheWarteschlange.java](https://github.com/bschlangaul/graph/algorithmen/BreitenSucheWarteschlange.java)

## Literatur

- [1] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 6. Graphen*. [https://www.studon.fau.de/file2635324\\_download.html](https://www.studon.fau.de/file2635324_download.html).
- [2] Uwe Schneider. *Taschenbuch der Informatik*. 7. Aufl. Hanser, 2012. ISBN: 9783446426382.
- [3] Wikipedia-Artikel „Breitensuche“. <https://de.wikipedia.org/wiki/Breitensuche>.
- [4] Wikipedia-Artikel „Tiefensuche“. <https://de.wikipedia.org/wiki/Tiefensuche>.