Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. Dr. h.c. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Übung 06: Informationstechnik (IT)

Marc Weber

Institut für Technik der Informationsverarbeitung (ITIV)

Teil 2: Sortier- & Suchalgorithmen

Inhalt: Übung 06 – Teil 2

1. • Sortieralgorithmen
2. • Suchalgorithmen

© Institut für Technik der Informationsverarbeitung (ITIV)

Algorithmen

- Genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten

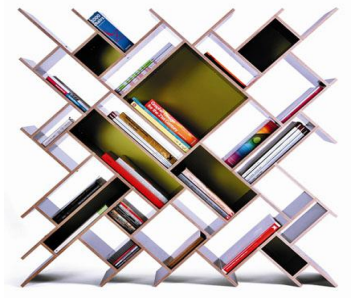
- Wichtige Problemstellungen
  - Sortieren
  - Suchen
  - Optimieren

© Institut für Technik der Informationsverarbeitung (ITIV)

Sortieren

- Problemstellung
  - Wie kann ich Datensätze sortieren / sortiert ausgeben?
  - Wie kann ich Personen nach Einkommen sortieren?
  - Wie kann ich Daten sortieren, um diese schneller zu verarbeiten?

- Lösung: Sortieralgorithmen
  - Sortieren eines Arrays oder einer Liste
  - Verschiedene Algorithmen
    - Unterschiedliche Geschwindigkeit, Rechen- und Speicherbedarf



© Institut für Technik der Informationsverarbeitung (ITIV)

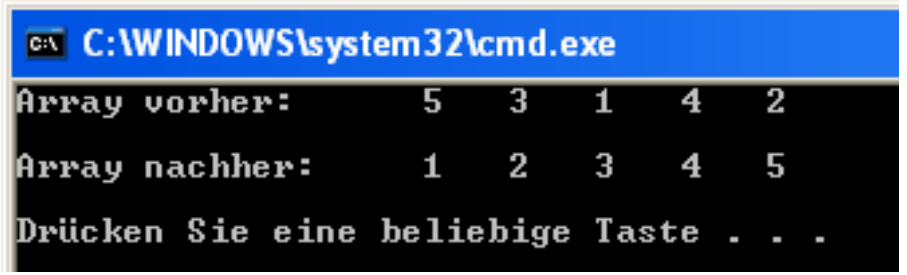
BubbleSort

- Einfacher Algorithmus – langsam, aber kaum zusätzlicher Speicherplatz

- Von Pseudocode zu C++ Quellcode

```
A = [5, 3, 1, 4, 2]
for i = 1 to länge[A] – 1
  do for j = 1 to länge[A] – i
    do if A[j] > A[j+1]
      then vertausche A[j] mit A[j+1]
```

```
int A[] = {5, 3, 1, 4, 2};
for( int i = 0; i < 4; i++ ) {
  for( int j = 0; j < 4 - i; j++ ) {
    if( A[j] > A[j+1] ) {
      int temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
  }
}
```



© Institut für Technik der Informationsverarbeitung (ITIV)

InsertionSort

- Sortieren durch Einfügen
  - Relativ einfacher Algorithmus

- Vorteile
  - Effizient bei kleinen & bei schon vorsortierten Eingabemengen
  - Einfache Implementierung
  - Stabil
  - Minimal im Speicherverbrauch

- Nachteile
  - Weniger effizient wie komplizierte Sortierverfahren

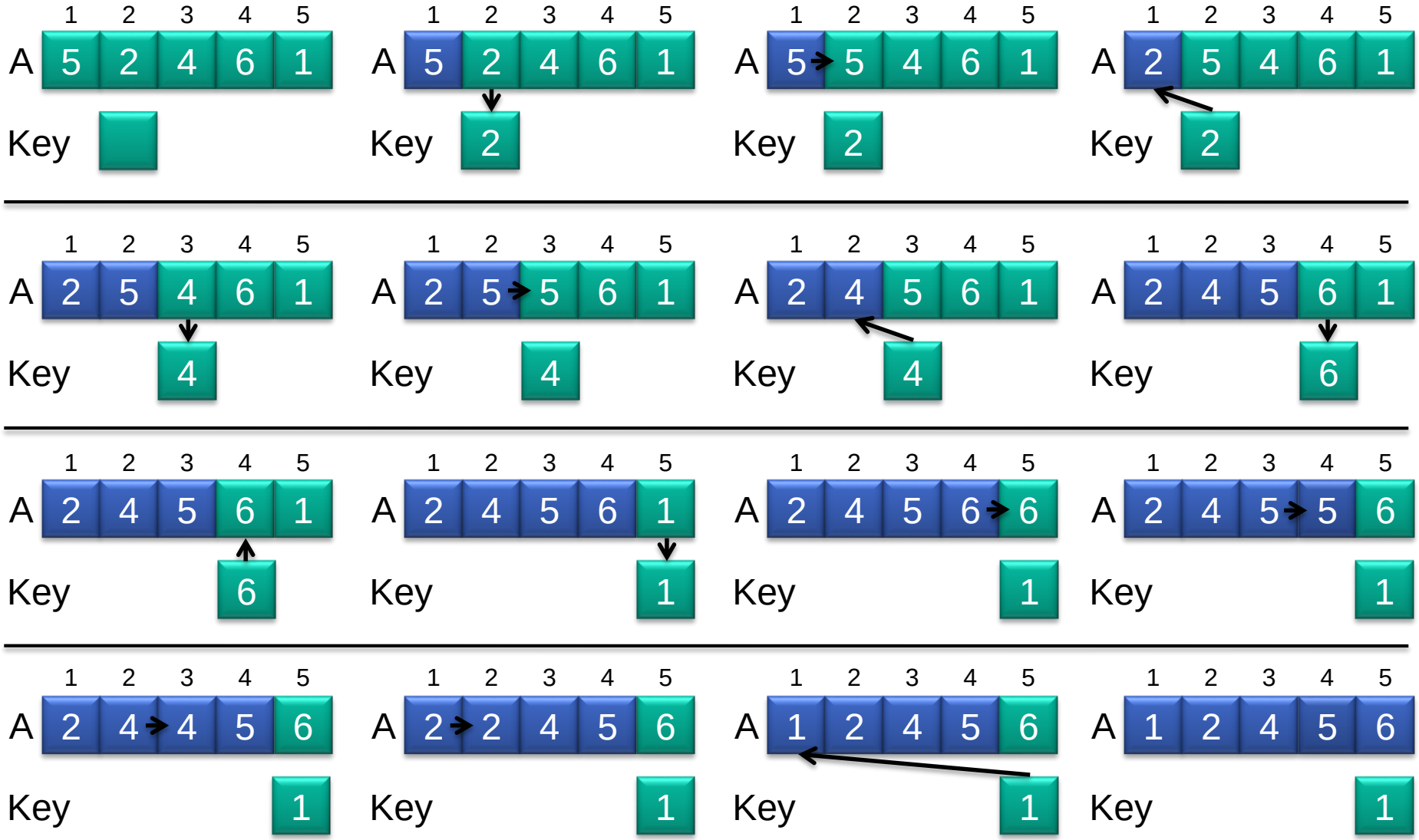
Link zu Demonstration:
http://einstein.informatik.uni-oldenburg.de/20907.html

InsertionSort

```
for ( j = 2 to length(A) ) do
    key = A[j]
    i = j - 1
    while ( i > 0 and A[i] > key ) do
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```


© Institut für Technik der Informationsverarbeitung (ITIV)

InsertionSort Ablauf



© Institut für Technik der Informationsverarbeitung (ITIV)

QuickSort

- Schneller, rekursiver, nicht-stabiler Sortieralgorithmus nach dem Prinzip „Teile und Herrsche“
- Ca. 1960 von C. Antony R. Hoare in seiner Grundform entwickelt und seitdem von vielen Forschern verbessert
- Verfügt über eine sehr kurze innere Schleife (was die Ausführungsgeschwindigkeit stark erhöht)
- Kommt ohne zusätzlichen Speicherplatz aus (abgesehen von dem Platz auf dem Aufruf-Stack für die Rekursion)
- Im Mittel schnellster Sortieralgorithmus $O(n \log_2 n)$ im schlechtesten Fall $O(n^2)$

© Institut für Technik der Informationsverarbeitung (ITIV)

QuickSort Prinzip

- Drei Schritte für das Sortieren eines Feldes **`A[p ... r]`**:

a) Teile: Zerlege das Feld **`A[p..r]`** in zwei (möglicherweise leere) Teilfelder **`A[p..q-1]`** und **`A[q+1..r]`**. Dabei soll jedes Element von **`A[p..q-1]`** kleiner oder gleich **`A[q]`** und jedes Element von **`A[q+1..r]`** größer als **`A[q]`** sein. **`A[q]`** ist dabei ein beliebiges Element des Feldes. (Somit sind alle Werte von **`A[p..q-1]`** auch kleiner wie die Werte in **`A[q+1..r]`**.)

b) Beherrsche: Sortiere die beiden Teilfelder **`A[p..q-1]`** und **`A[q+1..r]`** wieder durch rekursiven Aufruf von QuickSort (Beginn wieder beim Teilen), wenn die Teilfelder mehr als ein Element haben.

c) Verbinde: Da die Teilfelder in-place sortiert werden (in Feld selbst), ist keine Arbeit erforderlich, um sie zu verbinden. Das gesamte Feld **`A[p..r]`** ist nun sortiert.

© Institut für Technik der Informationsverarbeitung (ITIV)

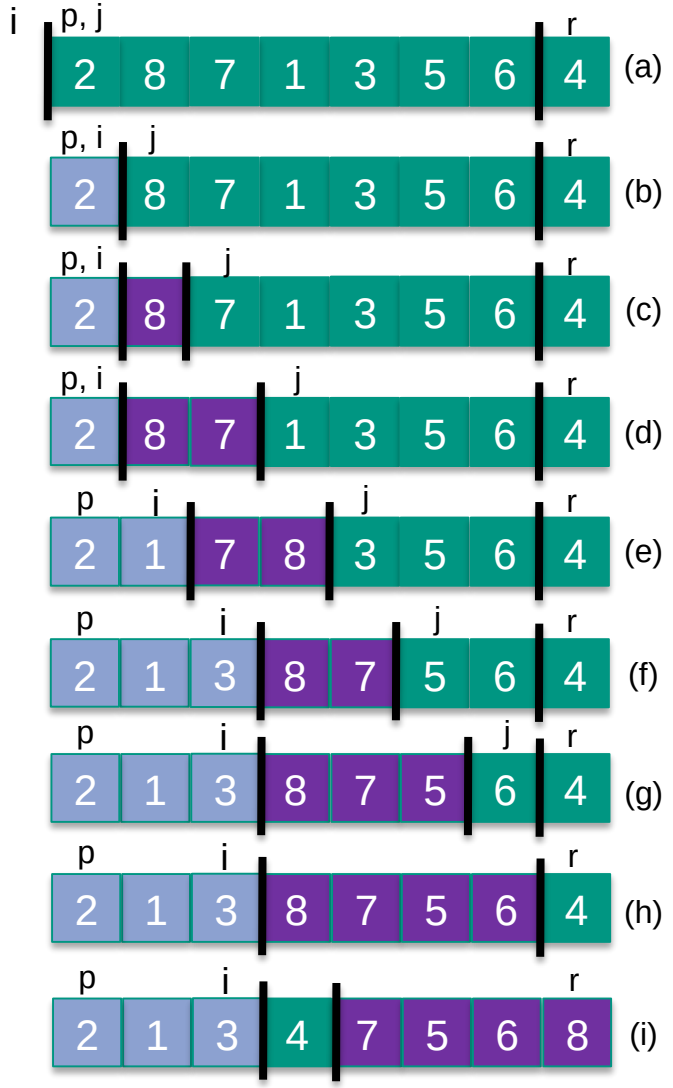
QuickSort Animation (siehe ILIAS)

```
Quicksort( A, 1, länge[A] )

Quicksort( A, p, r )
    if p < r
    then q = Partition( A, p, r )
      Quicksort( A, p, q – 1 )
      Quicksort( A, q + 1, r )

Partition( A, p, r )
      x = A[r]
      i = p - 1
      for j = p to r – 1
      do if A[j] <= x
        then i = i + 1
        vertausche A[i] mit A[j]
      vertausche A[i+1] mit A[r]
      return i+1
```

© Institut für Technik der Informationsverarbeitung (ITIV)

QuickSort Pseudocode

```
Quicksort( A, 1, länge[A] )

Quicksort( A, p, r )
        if p < r
        then q = Partition( A, p, r )
          Quicksort( A, p, q – 1 )
          Quicksort( A, q + 1, r )

Partition( A, p, r )
        x = A[r]
        i = p - 1
        for j = p to r – 1
        do if A[j] <= x
          then i = i + 1
          vertausche A[i] mit A[j]
        vertausche A[i+1] mit A[r]
        return i+1
```



© Institut für Technik der Informationsverarbeitung (ITIV)

QuickSort Partition Erklärung

- Die Arbeitsweise von Partition angewendet auf ein Beispielfeld. Hellblau schattierte Feldelemente befinden sich alle in der ersten Partition, in der die Werte nicht größer als x sind. Lila schattierte Elemente befinden sich in der zweiten Partition mit Werten, die größer als sind als x. Die grün schattierten Elemente sind noch nicht in eine der beiden ersten Partitionen eingefügt worden. Das letzte Element ist das Pivotelement.

  (a) Das Anfangsfeld und die gesetzten Parameter. Keines der Elemente ist in eine der beiden ersten Partitionen eingefügt worden.

  (b) Der Wert 2 wird „mit sich selbst vertauscht“ und in die Partition mit den kleineren Werten eingefügt.

  (c-d) Die Werte 8 und 7 werden der Partition mit den größeren Werten hinzugefügt.

  (e) Die Werte 1 und 8 werden vertauscht und die kleinere Partition wächst.

  (f) Die Werte 3 und 7 werden vertauscht und die kleinere Partition wächst.

  (g-h) Die größere Partition wächst, um die Elemente 5 und 6 aufzunehmen. Anschließend terminiert die Schleife.

  (i) In den Zeilen 7-8 wird das Pivotelement so eingefügt, dass es zwischen den beiden Partitionen liegt.

Link zu Demonstration: http://www.hermann-gruber.com/lehre/sorting/Quick/Quick.html

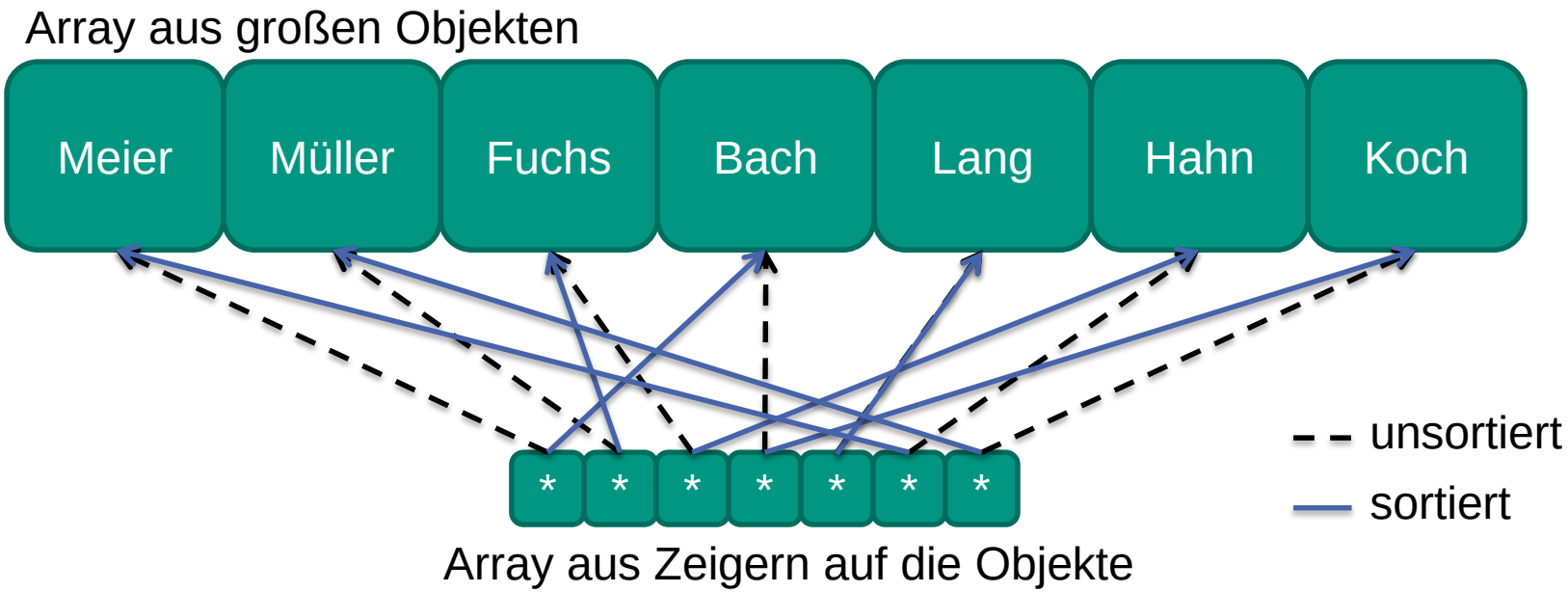
© Institut für Technik der Informationsverarbeitung (ITIV)

Sortieren und Zeiger

- Problemstellung
  - Wie kann ich große Objekte effizient und schnell sortieren?
  - Große Objekte = viele Attribute, wobei nach einem bestimmten Attribut aufsteigend oder absteigend sortiert werden soll
- Lösung: Sortieren von Zeigern auf Objekte
  - Man erstellt ein Array aus Zeigern, welche auf alle Objekte zeigen und sortiert nur diese Zeiger
- Vorteile
  - Wesentlich schnelleres sortieren bei großen Objekten, da nur Zeiger verschoben werden
  - Mit konstanten Zeiger können die Objekte vor einer ungewollten Veränderung geschützt werden (`const int* arr`)
- Nachteile
  - Es wird zusätzlicher Speicherplatz für die Zeiger benötigt
  - Komplexer in der Programmierung

© Institut für Technik der Informationsverarbeitung (ITIV)

Sortieren von Zeigern – Prinzip



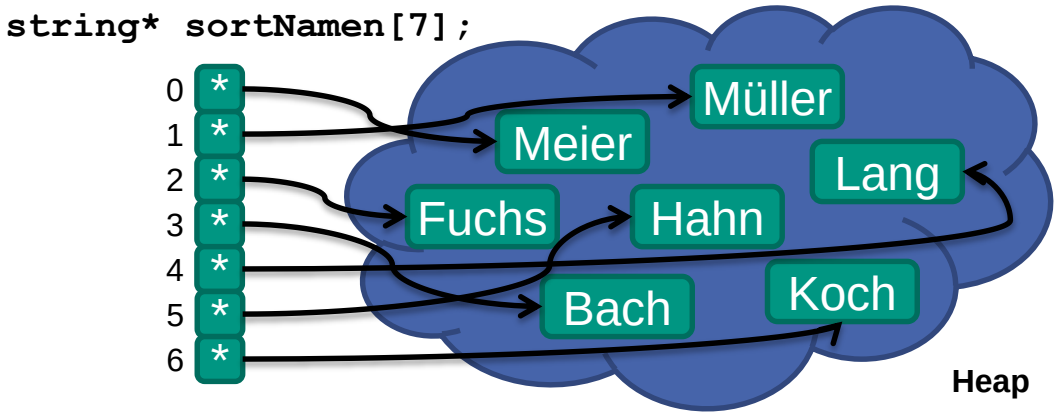
- Nur Veränderung der Zeiger, anstatt die großen Objekte im Array zu verschieben
- Für einzelne dynamisch erzeugte Objekte auf dem Heap ist kein anderes Sortierverfahren anwendbar

© Institut für Technik der Informationsverarbeitung (ITIV)

Beispiel: Sortieren dynamischer Objekte (1)

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class NamenSortieren {
private:
  string* sortNamen[7];
  int laenge;

public:
  void sortieren();
  void erzeugen();
  void ausgeben();
};
```

```
void NamenSortieren::erzeugen() {
  sortNamen[0] = new string( "Meier" );
  sortNamen[1] = new string( "Mueller" );
  sortNamen[2] = new string( "Fuchs" );
  sortNamen[3] = new string( "Bach" );
  sortNamen[4] = new string( "Lang" );
  sortNamen[5] = new string( "Hahn" );
  sortNamen[6] = new string( "Koch" );
  laenge = 7;
}
```



© Institut für Technik der Informationsverarbeitung (ITIV)

Beispiel: Sortieren dynamischer Objekte (2)

```
void NamenSortieren::sortieren() {
  string* temp = NULL;

  for( int i = 0; i < laenge - 1; i++ ) {
    for( int j = laenge - 1; j >= i + 1; j-- ) {
      if( *sortNamen[j] < *sortNamen[j-1] ) {
        temp = sortNamen[j];
        sortNamen[j] = sortNamen[j-1];
        sortNamen[j-1] = temp;
      }
    }
  }
}

void NamenSortieren::ausgeben() {
  cout << left;
  for( int i = 0; i < laenge; i++ )
    cout << setw( 10 ) << *sortNamen[i];

  cout << endl;
}
```

BubbleSort-Algorithmus zum Sortieren

Vergleich des Inhalts, worauf das Element im Zeigerarray zeigt

Nur verändern der Zeiger im Zeigerarray (Adressen werden vertauscht)

Linksbündig ausgeben

Ausgabe der Namen über Dereferenzierung

```
C:\WINDOWS\system32\cmd.exe
Meier      Mueller    Fuchs      Bach       Lang       Hahn       Koch
Bach       Fuchs      Hahn       Koch       Lang       Meier      Mueller
Drücken Sie eine beliebige Taste . . .
```


© Institut für Technik der Informationsverarbeitung (ITIV)

- Algorithmen
- Sortieren
  - BubbleSort
  - InsertionSort
  - QuickSort
- Sortieren und Zeiger
- Sortieren dynamischer Objekte

© Institut für Technik der Informationsverarbeitung (ITIV)

Zwischenübung 01: Bewertung Alg.

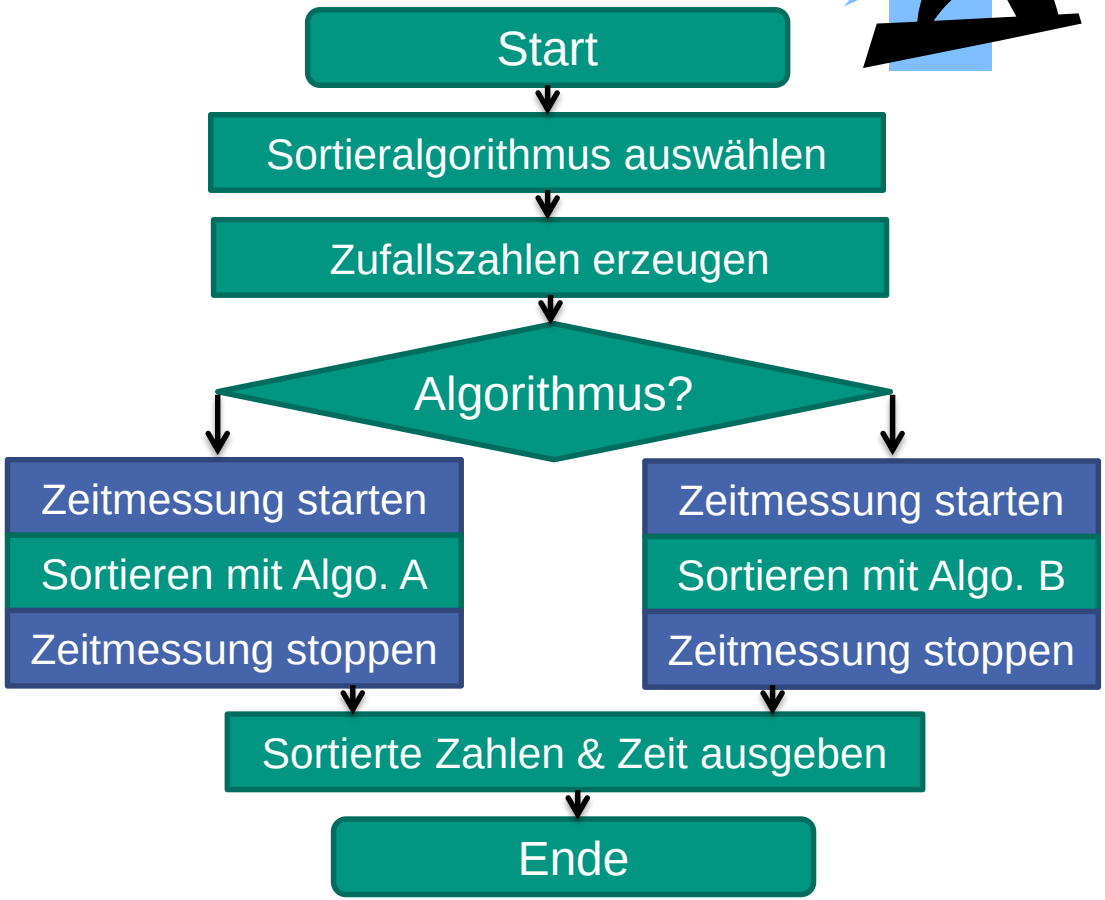
Situationsbeschreibung:

Zwei Forscher treffen sich und möchten sich über die Geschwindigkeit von verschiedenen Sortieralgorithmen austauschen.

Forscher A hat hierzu ein Testprogramm geschrieben, welches er Forscher B als Ablaufdiagramm vorstellt und anschließend zweimal ausführt, um seine Ergebnisse zu zeigen.

Daraufhin Forscher B zu Forscher A …

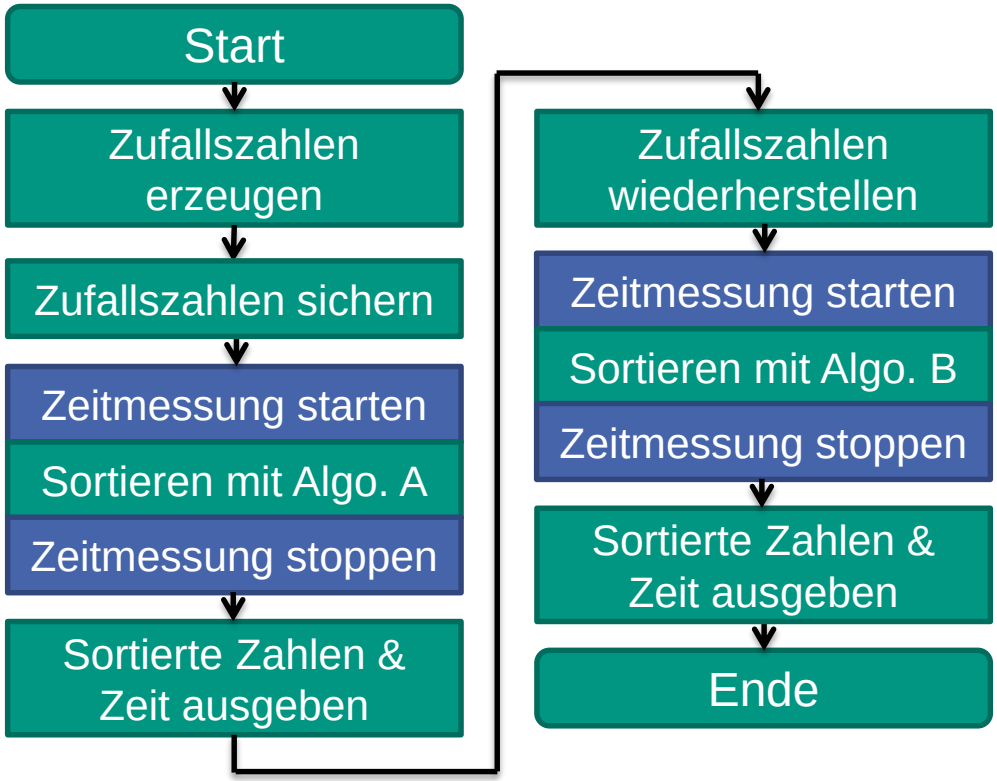
Link zu Demonstration: http://www.gf-webdesign.de/sortieralgorithmen/



© Institut für Technik der Informationsverarbeitung (ITIV)

Zwischenübung 01: Bewertung Alg. Lsg.

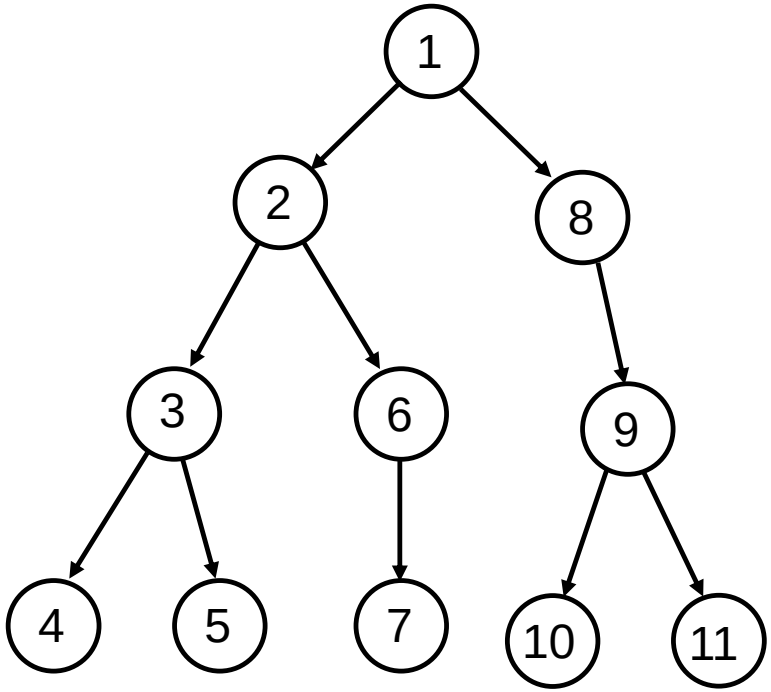
Daraufhin Forscher B zu Forscher A du kannst nicht zwei verschiede Algorithmen miteinander vergleichen, wenn du nicht die gleichen Vorbedingungen herstellst (gleiche Zufallszahlen). Am Besten du baust dein Programm so auf (siehe rechts).



© Institut für Technik der Informationsverarbeitung (ITIV)

Allgemeine Tiefensuche

- Gegeben ist ein Graph, sowie ein Anfangsknoten im Graphen, gesucht ist ein Knoten mit einer bestimmten Eigenschaft
- Besuche der Reihe nach alle Knoten, die über Kanten erreichbar sind
  - Also: Zunächst den ersten vom Anfangsknoten erreichbaren Knoten
  - *Dann*: Den ersten von diesem Knoten aus erreichbaren, und so weiter
  - Wenn kein Knoten mehr erreichbar ist, kehre zum letzten Punkt zurück, an dem es noch weitere Kanten gab und weiter wieder mit *Dann*
  - Wenn vom Anfangsknoten alle Kanten abgesucht wurden, ist der gesamte erreichbare Teil des Graphen abgesucht



© Institut für Technik der Informationsverarbeitung (ITIV)

Tiefensuche

- Die Tiefensuche (Depth First Search – DFS) exploriert den Graphen zuerst in die Tiefe, wenn möglich. Sie untersucht zuerst die Kanten des zuletzt entdeckten Knotens. Erst wenn diese alle abgearbeitet sind, wird zum letzten Knoten zurückgekehrt (eine Art Backtracking).

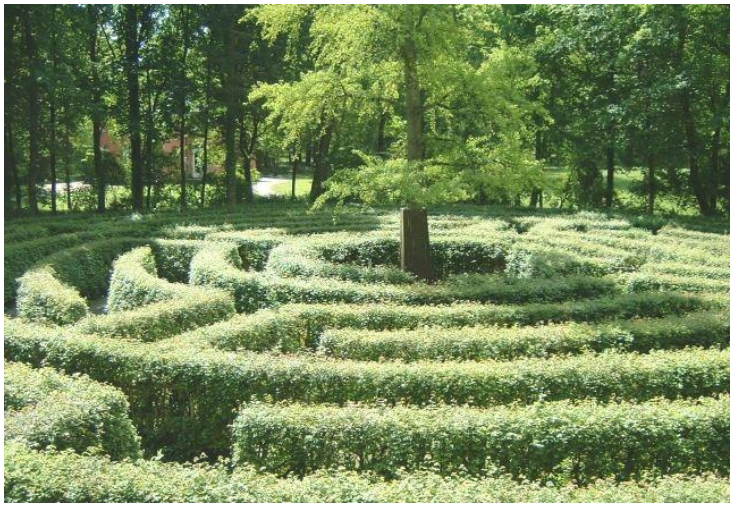
- DFS für einen geg. Graphen G = (V, E) und Startknoten S:
  - Verwendet eine Einfärbung der Knoten
    - weiß == noch nicht begonnen
    - grau == in Bearbeitung
    - schwarz == abgeschlossen

- Führt die Entdeckungszeit **`starttime[u]`** und die Bearbeitungsendzeit **`endtime[u]`** mit

© Institut für Technik der Informationsverarbeitung (ITIV)

Tiefensuche: Anwendung auf ein Labyrinth

- Aufgabe: Durchsuchen eines Labyrinths

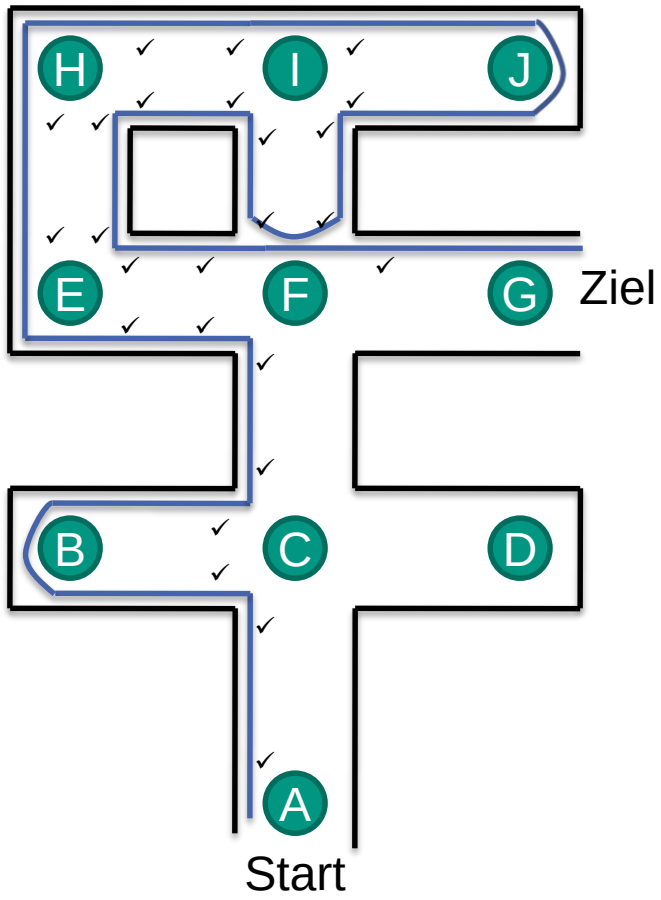


- Präzisierung der Aufgabe
  - Es soll sichergestellt werden, dass ein Weg vom Start zum Ziel gefunden wird, auch wenn das Labyrinth nicht bekannt ist
  - Die Sichtweite beträgt immer nur ein Feld in jede Richtung
  - Es soll verhindert werden, dass unbemerkt im Kreis gegangen wird
  - Es ist nicht notwendig, den kürzesten Weg zu finden

© Institut für Technik der Informationsverarbeitung (ITIV)

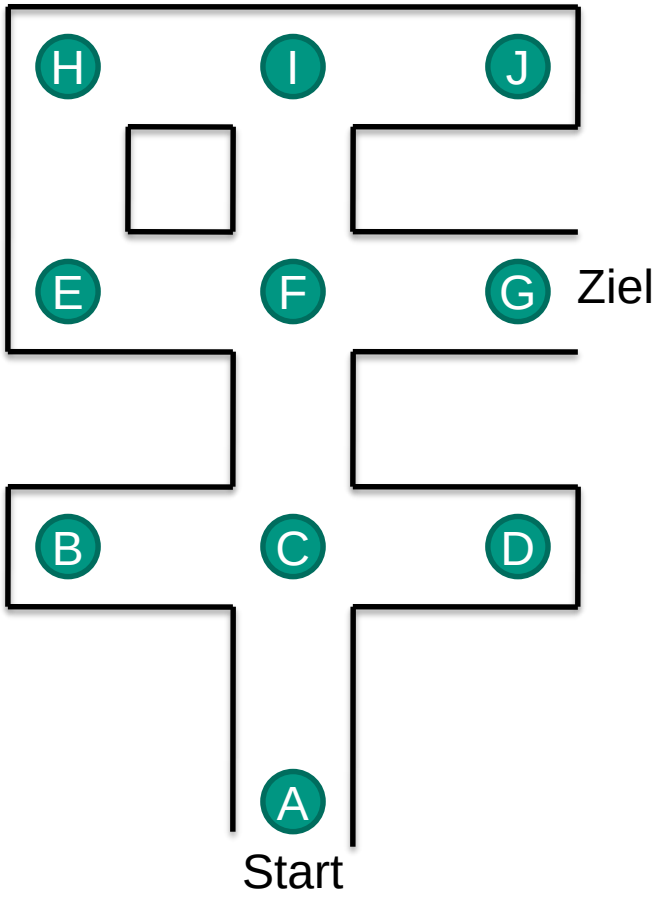
Umsetzung des Tiefensuch-Algorithmus (1)

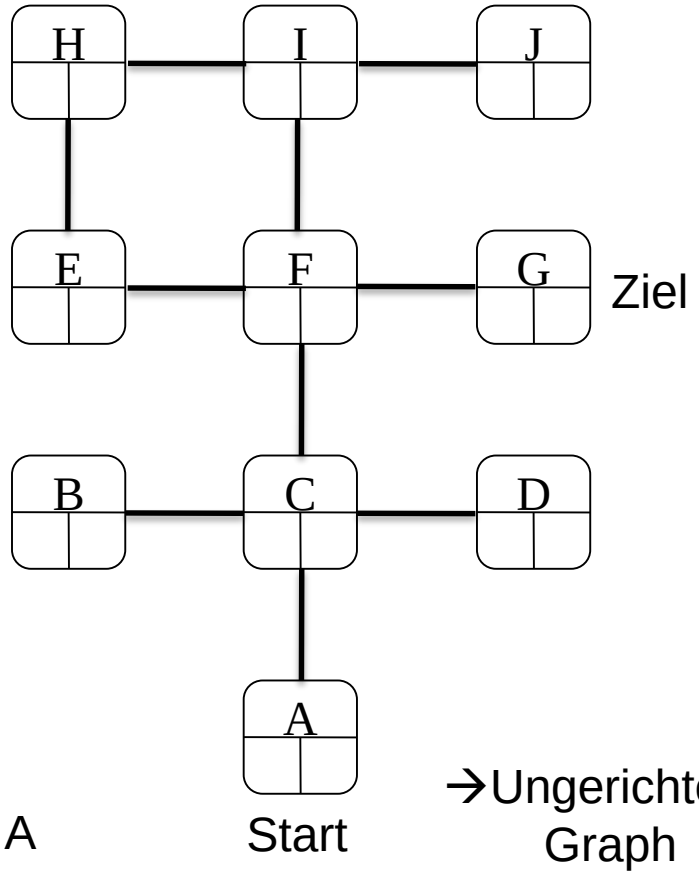
- Lösung mit Hilfe von Markierungen an den Kreuzungen
- Sackgasse: Umdrehen. (Knoten hat keinen Nachfolger)
- Kreuzung: Beim Betreten, Haken an die Wand malen (Entdeckungszeit)
  - Nicht im Kreis laufen! Hat der Gang, durch den man gekommen ist, eben seinen ersten Haken bekommen und sind weitere Haken an der Kreuzung vorhanden: Zweiten Haken malen und umdrehen (Knoten ist nicht weiß)
  - Sonst: Unerkundete Gänge suchen! Falls Gänge ohne Markierungen vorhanden: Davon den Ersten von links nehmen, Haken an die Wand malen (nächsten weißen Knoten untersuchen)
  - Sonst: Zurückgehen: Den Gang nehmen, der nur einen Haken hat (zurück zum vorherigen Knoten)



© Institut für Technik der Informationsverarbeitung (ITIV)

Umsetzung des Tiefensuch-Algorithmus (2)





→Ungerichteter Graph

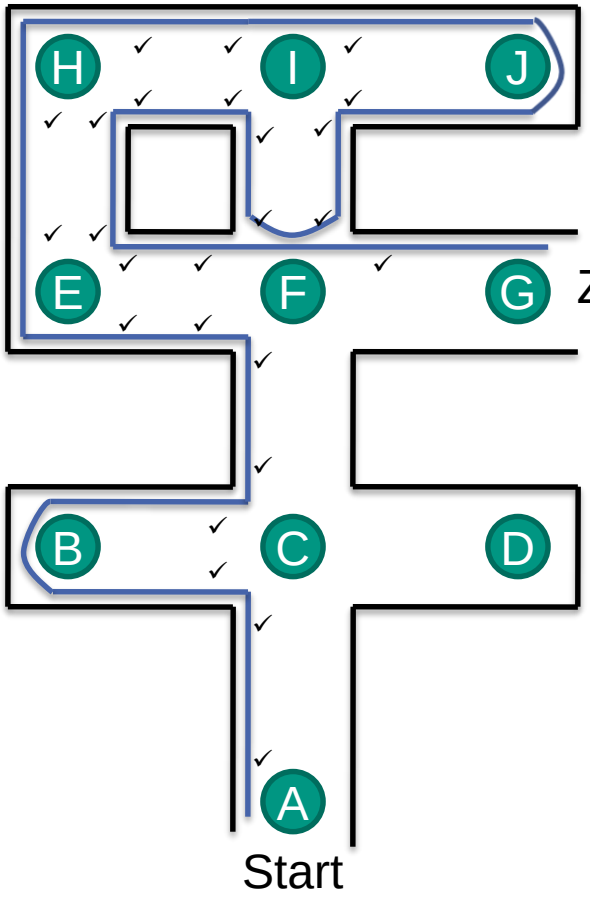
Startknoten A

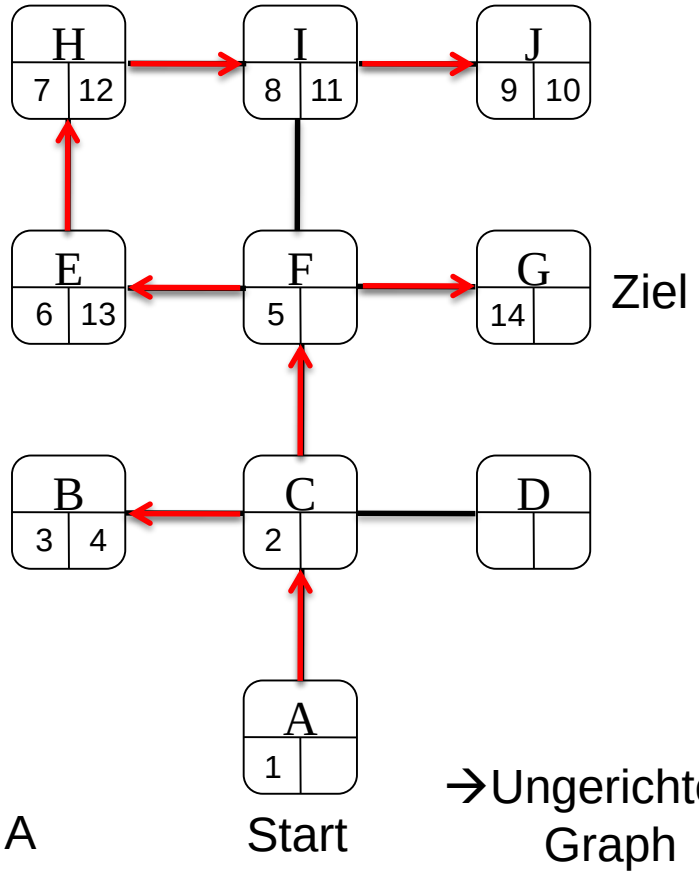
Gesuchter Knoten G

Gesucht: Ein Weg von A nach G

© Institut für Technik der Informationsverarbeitung (ITIV)

Umsetzung des Tiefensuch-Algorithmus (3)





→Ungerichteter Graph

Startknoten A

Gesuchter Knoten G

Gesucht: Ein Weg von A nach G

© Institut für Technik der Informationsverarbeitung (ITIV)

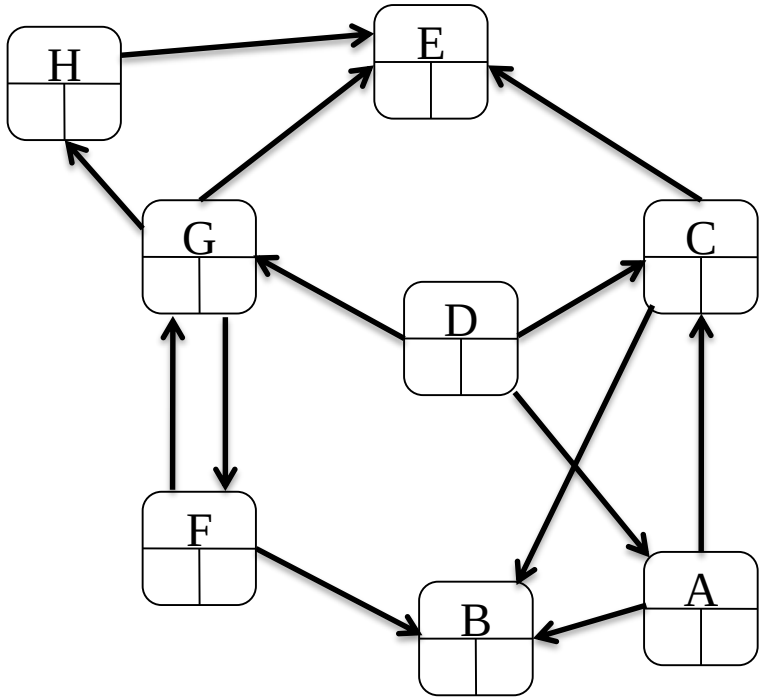
- Tiefensuche
  - Umsetzung

© Institut für Technik der Informationsverarbeitung (ITIV)

Zwischenübung 01: Tiefensuche

Wenden Sie den Pseudocode der Tiefensuche auf den folgenden Graphen an und geben Sie den daraus resultierenden Graphen mit der Entdeckungszeit und der Bearbeitungsendzeit an. Die Knoten werden dabei entsprechend ihrer alphabetischen Reigenfolge gefunden.

```
DepthFirstSearch( G )
  for alle Knoten u in G
  do farbe[u] = weiss
     vater[u] = NIL
  zeit = 0
  for alle Knoten u in G
  do if farbe[u] == weiss
     then DFS-VISIT( u )

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1
  startTime[u] = zeit
  for alle Knoten v aus Adj[u]
  do if farbe[v] == weiss
     then vater[v] = u
          DFS-VISIT( v )
  farbe[u] = schwarz; zeit = zeit + 1
  endTime[u] = zeit
```

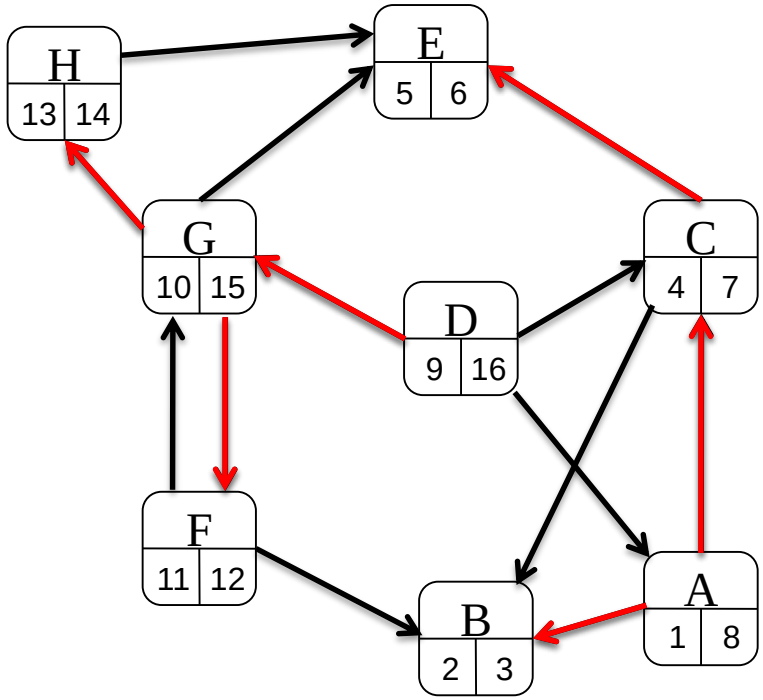


© Institut für Technik der Informationsverarbeitung (ITIV)

Zwischenübung 01: Tiefensuche Lsg. (1)

Wenden Sie den Pseudocode der Tiefensuche auf den folgenden Graphen an und geben Sie den daraus resultierenden Graphen mit der Entdeckungszeit und der Bearbeitungsendzeit an. Die Knoten werden dabei entsprechend ihrer alphabetischen Reigenfolge gefunden.

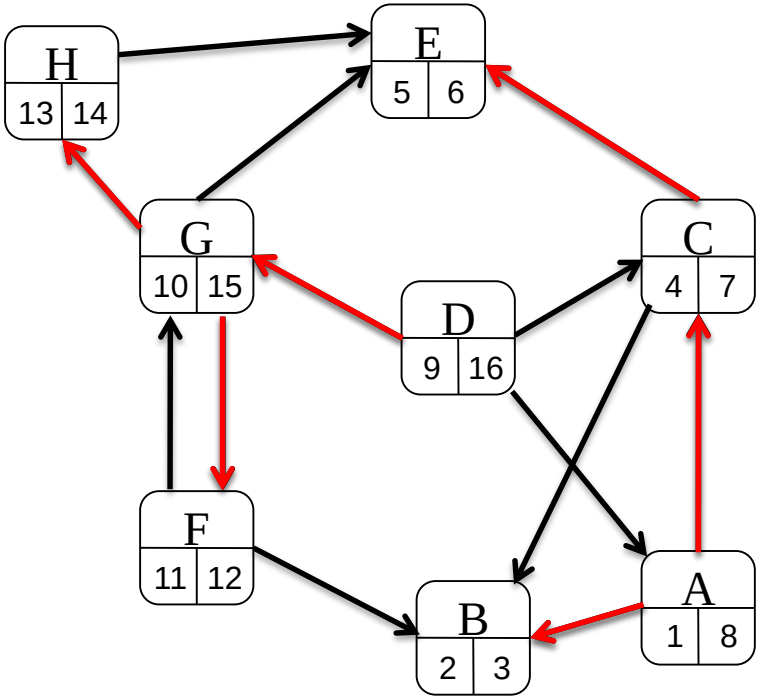
```
DepthFirstSearch( G )
  for alle Knoten u in G
  do farbe[u] = weiss
     vater[u] = NIL
  zeit = 0
  for alle Knoten u in G
  do if farbe[u] == weiss
     then DFS-VISIT( u )

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1
  startTime[u] = zeit
  for alle Knoten v aus Adj[u]
  do if farbe[v] == weiss
     then vater[v] = u
          DFS-VISIT( v )
  farbe[u] = schwarz; zeit = zeit + 1
  endTime[u] = zeit
```

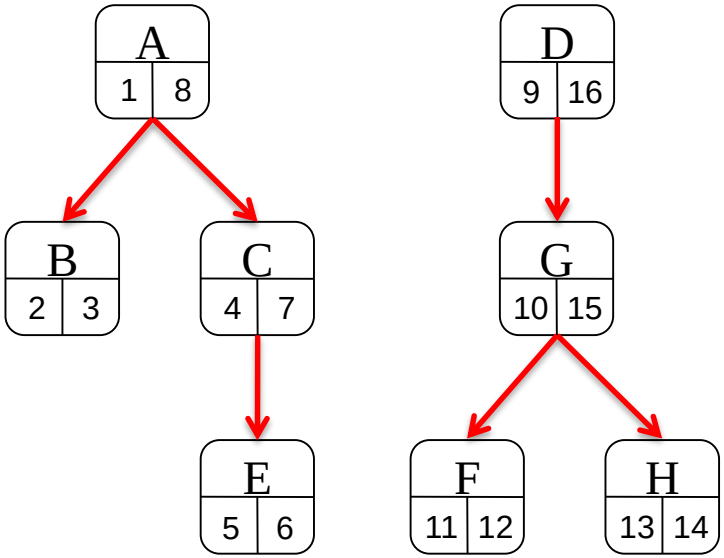


© Institut für Technik der Informationsverarbeitung (ITIV)

Zwischenübung 01: Tiefensuche Lsg. (2)

Wenden Sie den Pseudocode der Tiefensuche auf den folgenden Graphen an und geben Sie den daraus resultierenden Graphen mit der Entdeckungszeit und der Bearbeitungsendzeit an. Die Knoten werden dabei entsprechend ihrer alphabetischen Reigenfolge gefunden.



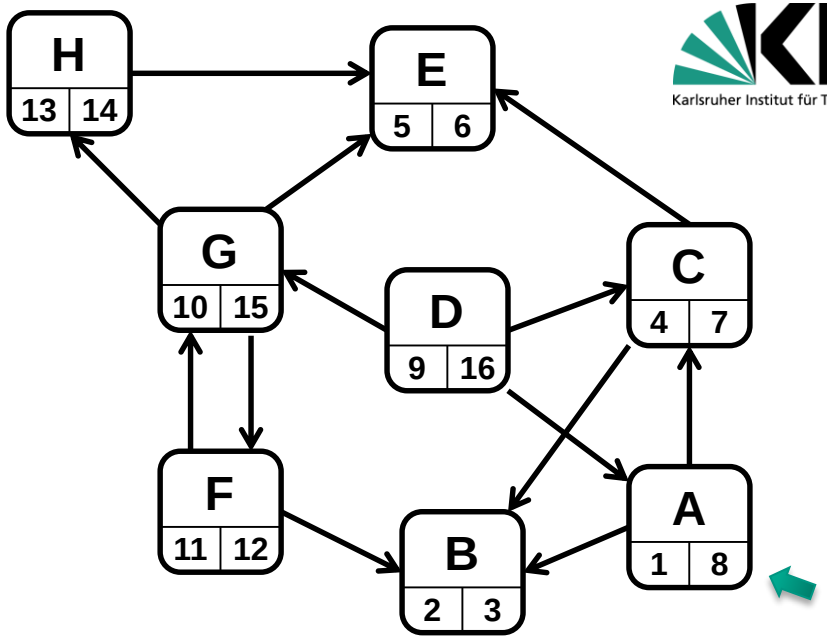


= Tiefensuchwald

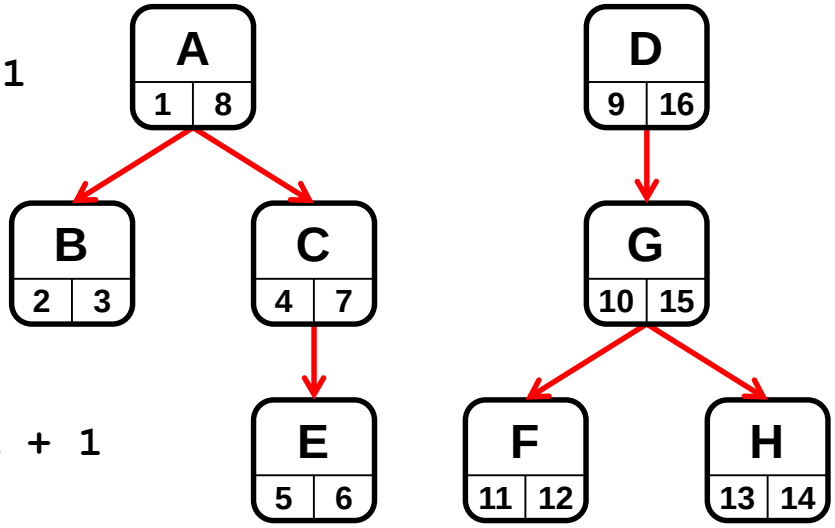
© Institut für Technik der Informationsverarbeitung (ITIV)

Tiefensuche Animation

```
DepthFirstSearch( G )
    for alle Knoten u in G
    do farbe[u] = weiss
       vater[u] = NIL
    zeit = 0
    for alle Knoten u in G
    do if farbe[u] == weiss
       then DFS-VISIT( u )
```




```
DFS-VISIT( u )
    farbe[u] = grau; zeit = zeit + 1
    startTime[u] = zeit
    for alle Knoten v aus Adj[u]
    do if farbe[v] == weiss
       then vater[v] = u
            DFS-VISIT( v )
    farbe[u] = schwarz; zeit = zeit + 1
    endTime[u] = zeit
```



© Institut für Technik der Informationsverarbeitung (ITIV)

Vielen Dank für Ihre Aufmerksamkeit

Marc Weber
Karlsruher Institut für Technology (KIT) – ITIV
marc.weber3@kit.edu

© Institut für Technik der Informationsverarbeitung (ITIV)