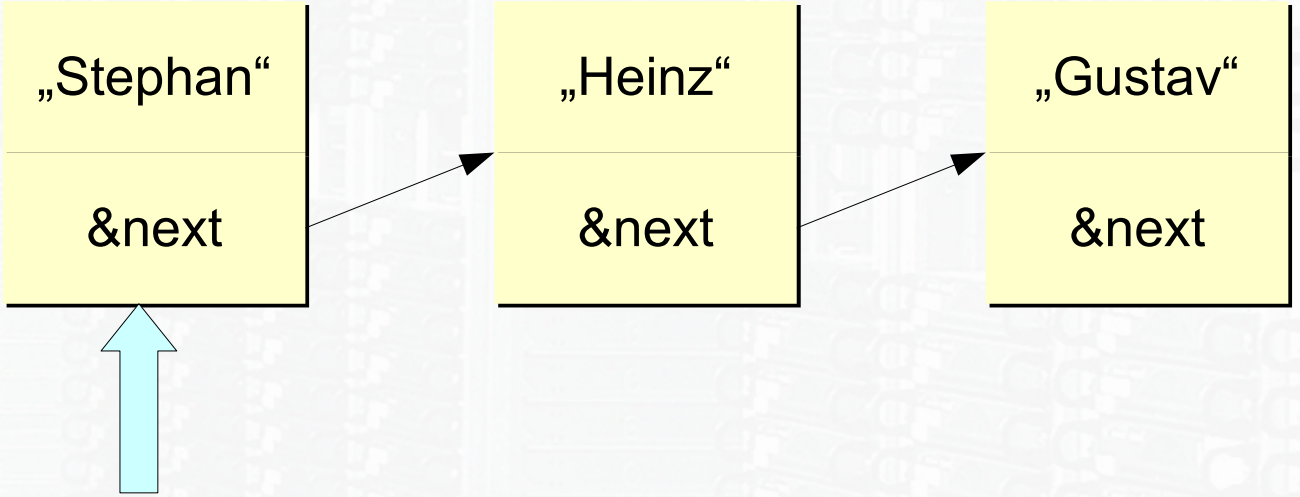
Datenstrukturen

einfach verkettete Liste

- speichert Daten in einer linearen Liste, in der jedes Element auf das nächste Element zeigt
- Jeder Knoten der Liste enthält beliebige Daten und einen Zeiger auf den nächsten Knoten in der Liste

- Data
- pointer



Stephan Alt
stephan.alt@rwth-aachen.de

Datenstrukturen

einfach verkettete Liste

- Die Liste selbst ist definiert über seinen ersten und seinen letzten Knoten (der letzte Knoten zeigt i.A. auf NULL)

```
typedef struct node {
      int data;                  // Datentyp von data beliebig
      struct node* next;         // Zeiger auf den nächsten Knoten
} Node, *nodeptr;
```

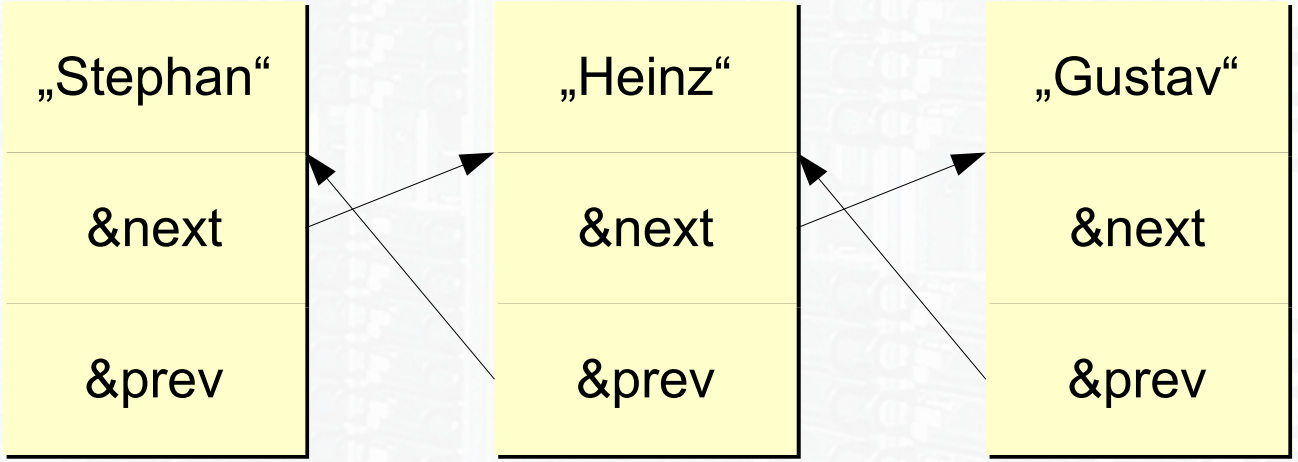
```
typedef struct list {
      struct node* first;        // Zeiger auf den ersten Knoten
      struct node* last;         // Zeiger auf den letzten Knoten
} List, *listptr;
```

Datenstrukturen

doppelt verkettete Liste

- ähnlich wie die einfach verkettete Liste, jedoch kennt jeder Knoten seinen Nachfolger und seinen Vorgänger
- Jeder Knoten der Liste enthält beliebige Daten und einen Zeiger auf den nächsten Knoten in der Liste

- Data
- pointer
- pointer



Datenstrukturen

doppelt verkettete Liste

- auch hier sollte gelten: myListptr->first->prev == NULL und myListptr->last->next == NULL

```
typedef struct node {
      int data;                  // Datentyp von data beliebig
      struct node* next;         // Zeiger auf den nächsten Knoten
      struct node* prev;         // Zeiger auf den Vorgänger
} Node, *nodeptr;
```

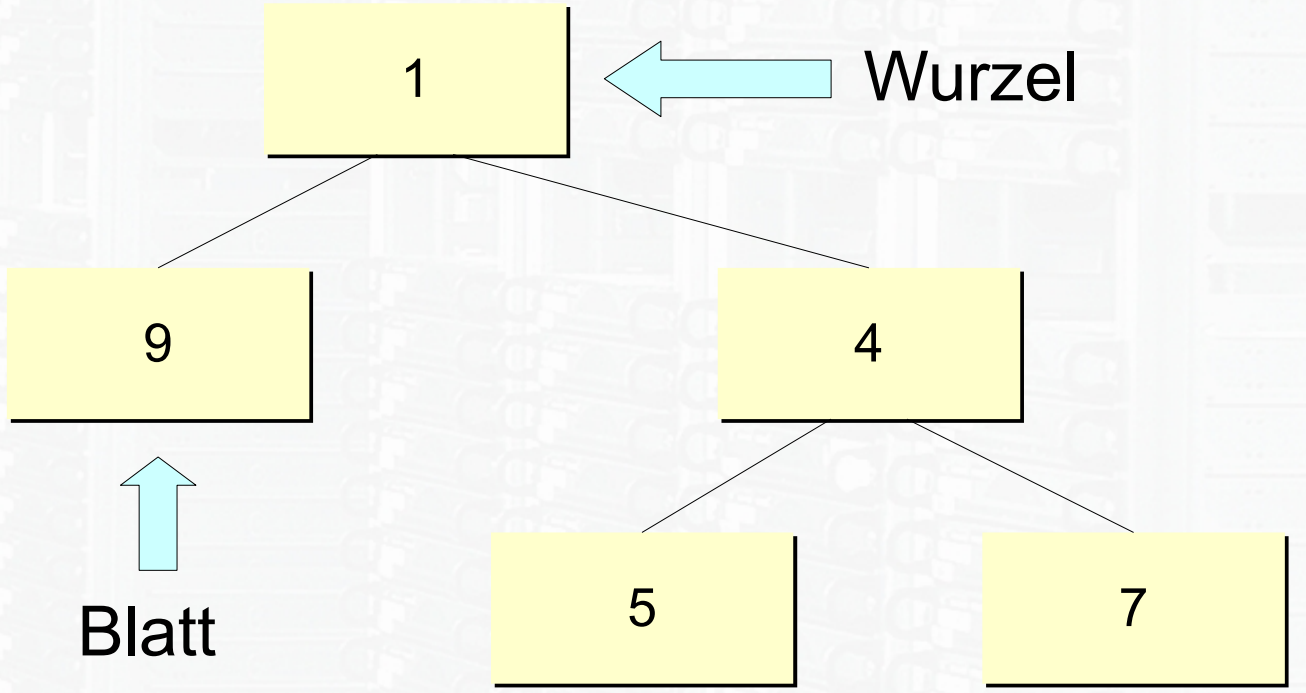
```
typedef struct list {
      struct node* first;        // Zeiger auf den ersten Knoten
      struct node* last;         // Zeiger auf den letzten Knoten
} List, *listptr;
```

Datenstrukturen

Bäume

- Jeder (Binär-)Baum ist eindeutig definiert aus seiner Wurzel und seinem linken/rechten Teilbaum
- Dabei kann ein (Teil-)Baum auch leer sein (keine Knoten enthalten)
- Knoten ohne Teilbäume heißen Blätter

```
typedef struct tree {
        int data;
        struct tree* left;
        struct tree* right;
} Tree, *treeptr;
```



Stephan Alt
stephan.alt@rwth-aachen.de

Datenstrukturen

Bäume

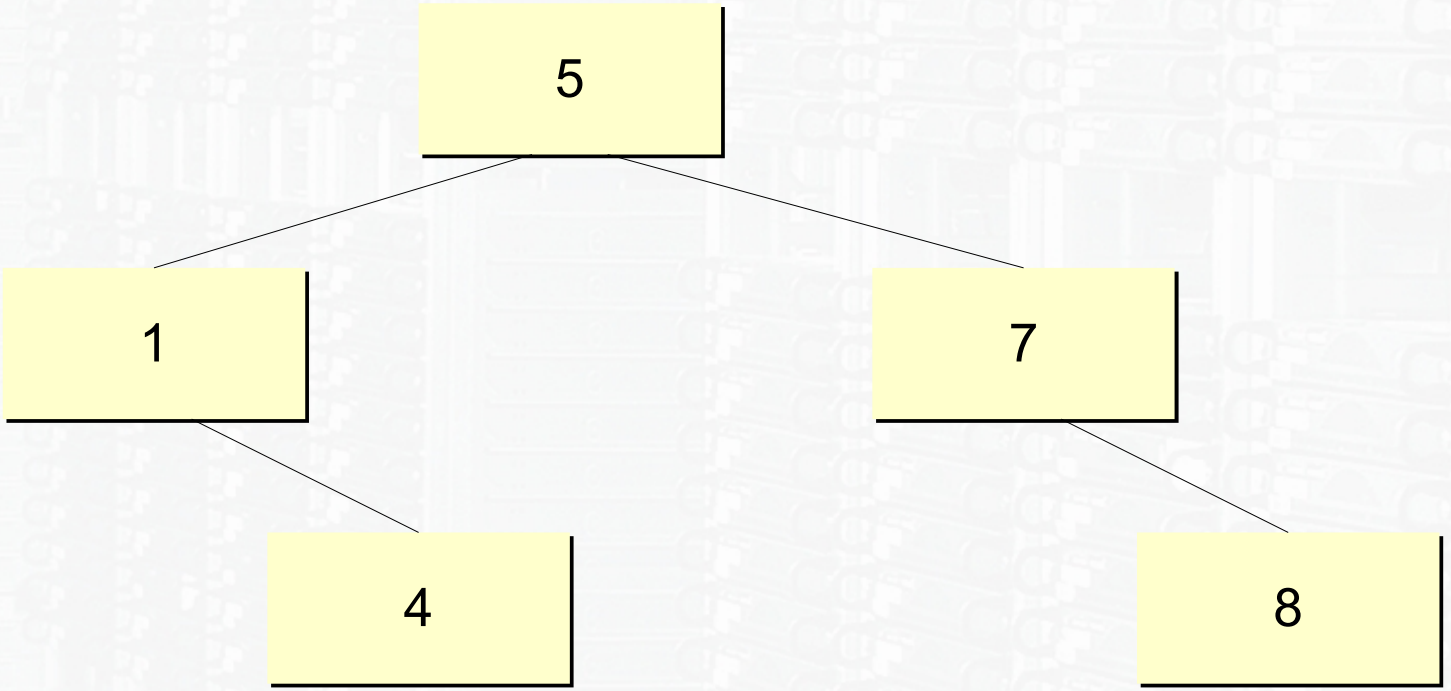
- Man unterscheidet bei Baumdurchläufen Pre- In- und Postorder
- Darunter versteht man den Zeitpunkt, zu dem man die Wurzel relativ zum linken / rechten Teilbaum betrachtet

Preorder	Inorder	Postorder
Wurzel linker Teilbaum rechter Teilbaum	linker Teilbaum Wurzel rechter Teilbaum	linker Teilbaum rechter Teilbaum Wurzel

Datenstrukturen

Suchbäume

- An einen Suchbaum sind die folgenden Bedingungen gestellt
- Alle Elemente des linken Teilbaums sind kleiner als die Wurzel, alle Elemente des rechten Teilbaums sind größer als die Wurzel
- linker und rechter Teilbaum sind selbst Suchbäume

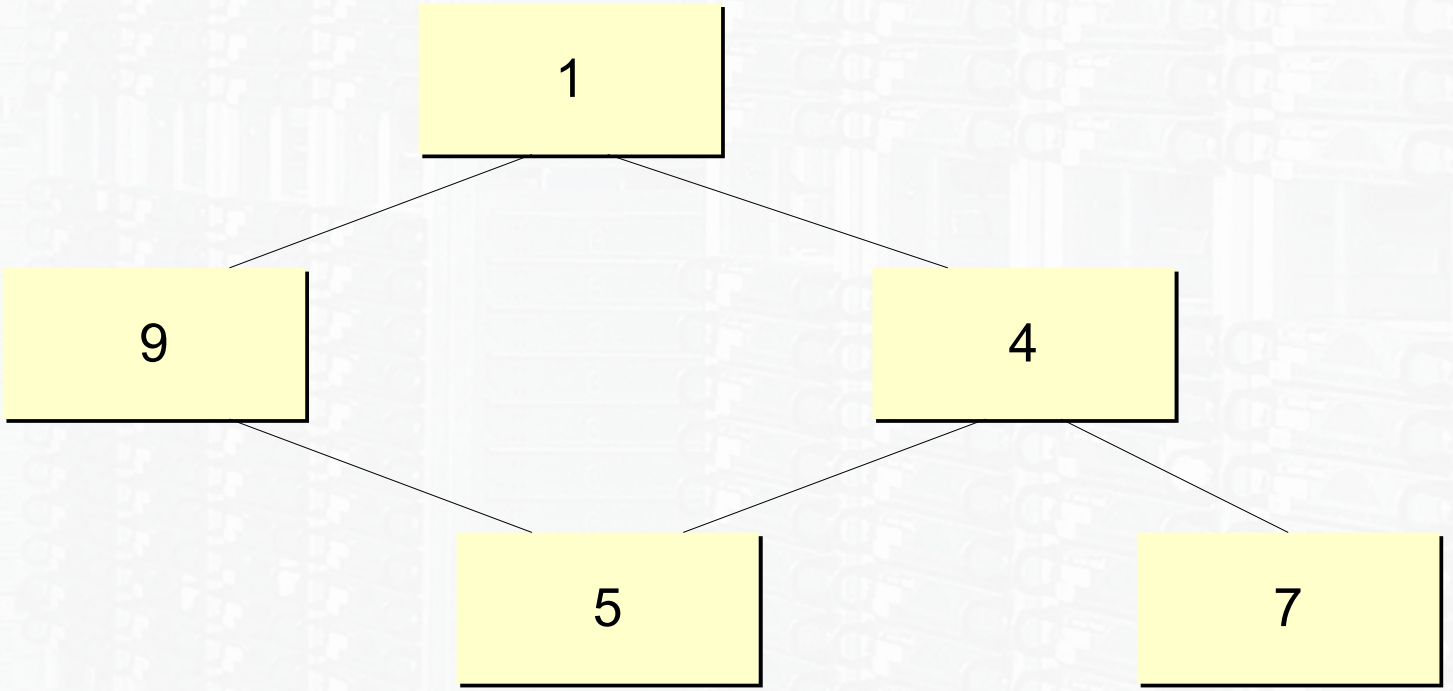


Stephan Alt
stephan.alt@rwth-aachen.de

Datenstrukturen

Graphen

- Jeder Graph besteht aus einer Menge V an Knoten und einer Menge $E \subseteq V \times V$ an Kanten
- Bäume, Listen etc sind alles spezielle Graphen



Stephan Alt
stephan.alt@rwth-aachen.de

Datenstrukturen

Graphen

- bei einem zusammenhängenden Graphen kann jeder Knoten über mindestens einen Pfad erreicht werden
- bei einem zyklenfreien Graphen gibt es höchstens einen Pfad von jedem Knoten zu Knoten
- jeder zusammenhängende und zyklenfreie Graph besitzt also genau einen Pfad von Knoten zu Knoten und ist somit immer ein Baum (jedoch nicht zwangsweise ein Binärbaum)

Graphdurchläufe

- Man unterscheidet DFS (= Depth First Search) und BFS (= Breadth First Search

Datenstrukturen

Graphen

- Die Menge der Kanten kann man auf verschiedene Weise darstellen
- Die Adjazenzmatrix speichert an seiner ij-ten Stelle den Wert der Kante von Knoten i zu j
  - Speicherverbrauch $O(V^2)$ Adjazenztest in O(1)
  - bei ungerichteten Graphen symmetrisch zur Hauptdiagonalen
- Bei der Speicherung mithilfe von Adjazenzlisten besitzt jeder Knoten eine Liste von Knoten, die von dort erreicht werden können
  - Speicherverbrauch O(E) Adjazenztest im worst case O(E)

Datenstrukturen

Dijkstra-Algorithmus (Bestimmung eines kürzesten Pfads)

- drei Mengen
  - gelbe (Knoten-)Menge: Menge der noch zu betrachtenden Knoten
  - grüne (Knoten-)Menge: Menge der bereits betrachteten Knoten
  - rote (Kanten-)Menge: Menge der günstigen Kanten
- schiebe den naheliegensten Knoten aus der gelben Menge in die grüne Menge und betrachte alle von ihm ausgehenden Kanten
  - ist der Zielknoten noch nie betrachtet worden, füge die Kante dorthin zur roten Menge hinzu, füge den Zielknoten zur gelben Menge hinzu und speichere den Abstand
  - ist der Zielknoten schon in der gelbe Menge und der neue Weg kürzer sein, so muss die alte rote Kante dorthin gelöscht werden und die neue Kante zur roten Menge hinzugefügt werden; der neue Abstand wird gespeichert

Datenstrukturen

Kruskal-Algorithmus
(Bestimmung eines minimalen / maximalen Spannbaums)

- zwei Kantenmengen
  - T-Menge: Menge der bereits als gut befundenen Kanten
  - L-Menge: Menge der noch zu betrachtenden Kanten (diese Menge muss sortiert sein)
- Entnehme das erste Element aus der L-Menge, überprüfe ob diese Kante vereinigt mit der T-Menge einen zyklenfreien Graphen aufspannen würde
  - Ja – füge das Element zur T-Menge hinzu
  - Nein – das Element wird fallen gelassen

Speicherabbilder

- relevant sind drei verschiedene Speicher
  - static storage: Speicherung von globalen Variablen
  - Stack: Speicherung lokaler Variablen und Rücksprungadressen
  - Heap: Speicherung aller Variablen, die mit malloc aufgerufen wurden
- Pointer speichern die Adresse einer anderen Variablen (werden meist durch einen Pfeil im Speicherabbild verdeutlicht)
- für Speicherabbilder relevante Informationen sind Adresse, Inhalt und Datentyp des Speichers (dabei kennt man nicht immer alle Informationen)